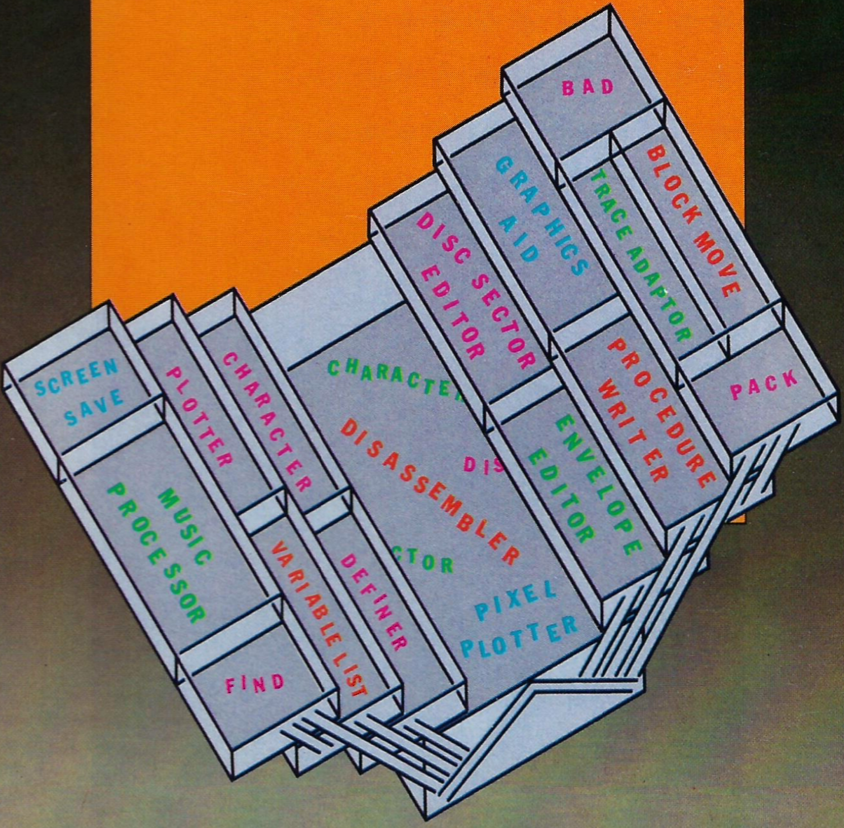
Pan

PERSONAL
COMPUTER
COMPUTER NEWS LIBRARY

JEFF AUGHTON

INVALUABLE UTILITIES for the BBC MICRO


BAD
GRAPHICS AID
TRACE ADAPTOR
BLOCK MOVE
DISC SECTOR EDITOR
PROCEDURE WRITER
PACK
ENVELOPE EDITOR
SCREEN SAVE
PLOTTER
CHARACTER
CHARACTE
DISASSEMBLER
DIS
MUSIC PROCESSOR
VARIABLE LIST
DEFINER
CTOR
PIXEL PLOTTER
FIND

Pan/Personal Computer News
Computer Library

Jeff Aughton

Invaluable Utilities for the BBC Micro

Pan Books London and Sydney

First published 1983 by Pan Books Ltd,
Cavaye Place, London SW10 9PG
in association with Personal Computer News
9 8 7 6 5 4 3 2 1
© Jeff Aughton 1984
ISBN 0 330 28480 0
Photoset by Parker Typesetting Service, Leicester
Printed and bound in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk

This book is sold subject to the condition that it shall not,
by way of trade or otherwise, be lent, re-sold,
hired out or otherwise circulated without the publisher's prior consent
in any form of binding or cover other than that
in which it is published and without a similar condition including
this condition being imposed on the subsequent purchaser

To my mother and father

Contents

Introduction

Section 1: Basic utilities
- Utility 1: Scrolling memory editor
- Utility 2: Single key memory display
- Utility 3: Printer memory dump
- Utility 4: Disassembler
- Utility 5: Block move
- Utility 6: String search

Section 2: BASIC utilities
- Utility 7: Highlight end-of-line spaces
- Utility 8: Remove end-of-line spaces
- Utility 9: Pack
- Utility 10: Bad
- Utility 11: No-colon LISTer
- Utility 12: Find
- Utility 13: Replace
- Utility 14: MODE 7 PROC writer
- Utility 15: New TRACE/BASIC single step
- Utility 16: Symbol table

Section 3: Sound and graphics
- Utility 17: ENVELOPE editor
- Utility 18: Music processor
- Utility 19: Character definer
- Utility 20: Screen save
- Utility 21: MODE 2 character creator
- Utility 22: MODE 2 character plotter
- Utility 23: Graphics aid
- Utility 24: MODE 7 pixel plotter

Section 4: Miscellaneous utilities
- Utility 25: Display user keys
- Utility 26: Printer screen dump
- Utility 27: String sort
- Utility 28: Universal input routine
- Utility 29: Date conversion PROCedure
- Utility 30: Disc sector editor
- Utility 31: Multiple precision arithmetic

Introduction

This book contains some thirty utility programs for the BBC microcomputer. As well as the standard utilities familiar to most micro users, we include some original programs that you will not have seen before.

Everyone, from the youngest beginner to the most hardened veteran, will occasionally need to use a utility program and this book is intended to be a sourcebook – a kind of programmer's 'toolkit' – of such facilities. Thus, whatever your level of experience, you should find something of interest in the following pages.

Before getting down to the main business of the book, let's begin by explaining precisely what we meant by the term 'utility'.

A utility is something useful – in computer terms it refers to any piece of code (not necessarily a program) that may help you as you go about the business of using your computer. The key word here is 'using', which implies your active involvement in a creative process (as opposed to the passive role of a participant in a video game). You may find games interesting and enjoyable but you should ask yourself who is running the show. . .!

Utilities are primarily thought of as programming aids, but the truth of the matter is that they have a part to play in any activity which involves computers. Cataloguing files, listing programs, designing envelopes – these are all areas in which utilities can help because they all use the computer in some way. In their most basic forms, the first two processes are served by the commands *CAT and LIST respectively; a later program will attend to the third.

You should already be familiar with the concept of utilities, because the BBC Micro has a number of built-in facilities – DELETE and RENUMBER are two good examples. These are not BASIC statements, they are part of the machine, compliments of the manufacturer, to be used as and when you see fit. If you do not write programs, then RENUMBER is no use to you. However, if you do, this utility (for that's what it is) will help you tidy them up and, more usefully, open up gaps for the insertion of additional lines.

DELETE illustrates the fact that utilities need not do anything esoteric. It is a simple matter to delete lines from a BASIC program yourself but this command allows you to remove a whole block of lines at one go, saving you time and trouble.

Notice the distinction here: PRINT and DELETE are two very different instructions. The first is an indispensable BASIC keyword (strictly speaking, hardly anything is indispensable, but that argument can get very subtle and is ultimately pointless), while the second should be regarded as a gift from Acorn. Actually, we BBC owners have been rather spoiled by such gifts, but there's always room for a few more – everyone likes presents – and that's what this book is about.

How many more things like DELETE do we need? Well, the answer really depends on what you want to do with your Beeb. Certainly, as your programming skills develop you will want to spend less time on chores (like typing in your own line numbers) and more time on the creative side of designing your program. Even if your computing activities are restricted to copying programs, then you will reach a point where you think 'If only I could let the computer get on with this while I. . .' – what you need here is a utility, and you will have to write it yourself unless one already exists. Quite a few of the routines in this book came about in this way.

About this book

The programs in this book have been written with three aims:

i) To help you write your own programs more easily and to save you time and effort while you are doing so. During the development of a program, situations arise that are not catered for by built-in utilities like TRACE and RENUMBER. A good example of this is the tenacious 'Bad program' error which, as you will know if you have suffered it, is quite difficult to shake off. As a result, this book contains a utility to deal with it.

Some utilities provides diagnostic information while others help you to use existing BASIC commands more efficiently (e.g. ENVELOPE), but they all share the common aim of making your programming life simpler. You will *not* find a utility to help you improve on your Space Invaders top score!.

ii) To provide you with some pre-coded procedures ready to slot into your own programs.

Some routines are so versatile that they crop up time after time in various forms, often in programs that would otherwise have very little in common. As an example of this type of utility we have a string sort routine that can be copied into any of your programs that needs to do some sorting – quite a common requirement. The distinction between these routines and those in (i) is very fine (especially in the case of sorts – they are almost always regarded as utilities), but we thought that we should raise the point before you did.

If you take the view that a utility is any program that saves you time while you are using your computer, then these procedures definitely qualify.

iii) To illustrate some interesting programming ideas.

Because of the wide scope of the programs, a variety of programming techniques have been employed to get the routines up and running. Although this is not a teaching book, the programs have been developed along the following lines: here's a problem, here's a possible solution, now here's a program to implement that solution. In this way the full background story to the program is given in addition to the coding itself. Even if you do not type in any of the programs you should find the listings interesting and the methods used will hopefully stimulate a few ideas of your own.

Although we shall make the point more forcefully later in the text, we would like to make it clear that these programs have been coded to be as useful and as versatile as possible. Being utilities, the emphasis had to be on practicality. This said, we also considered it essential that you should understand them enought to be able to amend them to suit your own purposes if they do not quite agree with ours. None of the coding has been (deliberately) included for effect – if it's there, then it's probably because it's the best thing we could think of consistent with the principles previously outlined.

Who needs utilities?

If you write programs, or even just amend them, then you do. Obviously not everyone will want to use all the utilities contained in this volume (you must be pretty prolific, or accident prone if you do), but we hope that all programmers will find something to interest them. Most of the routines are open to expansion in a number of ways (some of these enhancements are suggested), and even if you can't find the routine that precisely answers your particular need, there's a good chance that there's something close to it that can be adapted without too much trouble. We've tried to adopt the approach that all programs tend to get messed around with in some way and guidelines for 'good messing' are included.

About utilities

In general, utilities are subject to most of the standard programming principles, although it should be noted that in certain respects the ground rules are somewhat different.

The single most important thing about any program is that it must work ('properly' – but that means different things to different people) and this is certainly true of a utility. Nothing else really matters. If it doesn't work (or worse still, if it works incorrectly), it's not a utility – it's a liability! The nature of utilities dictates that they are often the sacrificial lambs for larger, more complicated programs which must be protected at all costs. If your utility has a fit and erases the host program, it hardly matters how well it has been written or whether the variable names are rigorously appropriate.

At this point the requirements for utilities and other programs diverge (actually the distinction is rather arbitrary – the 'principles of good programming' are suitably vague, and most of the time amount to basic common sense). A utility has to be easy to use. In its role as a time-saver, it should take the minimum amount of time to run – taking both human and computer time into account. Operational simplicity is very important, but satisfying this requirement can have an adverse effect on a feature that is probably more important – the size of the program. Many of these routines have to load into the computer alongside other programs, or hide in out of the way places until they are needed, and so they should be as short as possible.

This preoccupation with condensed code is definitely contrary to accepted high-level language etiquette. All the standard battle-cries – 'use lots of REMS', 'spread your program out', 'long variable names only' – are an anathema to the utility programmer. For maximum versatility (that's what utilities are about, remember), the program should be as short as possible, and there are many ways of packing instructions into a few lines.

Unfortunately, highly condensed programs are not suitable for publication in a book, especially one that purports to explain how each program works and how it may be modified. This has naturally entailed a few compromises.

We're not big fans of REM statements and each program has about one – right at the beginning. Likewise, very long variable names ('distance-between-two-points%') are not very popular and most of ours are reasonably short, but hopefully easily understood once the program is studied in conjunction with the 'variables' section.

None of the programs contain any operating instructions. No self-respecting utility is going to waste space telling you how it works and you will be expected to know how to run the program. This may not appear 'user-friendly', but in fact the friendliest thing a utility can do for you is work efficiently, and this means omitting the instructions. Where you have a number of choices, you may be given a prompt showing the expected responses and full operating instructions are provided with the description of each routine.

Similarly, another apparent omission is the lack of comprehensive vetting facilities. Experienced programmers will smile to themselves at this point and recall that in an average program total vetting of all input data can generate half as much code as the rest of the program. This is right out for utilities. Obviously we can't allow potentially dangerous inputs as they may wreck the utility (or more seriously, another program resident in the computer), so a certain amount of vetting is necessary. However, it is simply not practical to try to trap every invalid input and, if you are really determined to crash one of these programs, then you can, but it is not likely to happen by accident. You may consider it to be a standing entry in the 'Extensions' section that comprehensive data vetting and error handling could be added to the program.

Having just dismissed two of the most sacred principles of decent programming, let's make the following observation: The programs in this book are for *you* to use. As the owner/operator you are hardly likely to deliberately wreck your own program, so most of the vetting has been omitted on the grounds that it is only necessary to trap simple errors. The same applies to instructions. After you have used the program a couple of times you would probably resent page upon page of flashing, double height, instructions that ultimately only waste valuable program space. The amount of memory saved by the omission of these two items is both useful and necessary.

As we said, the programs presented here represent a compromise and hopefully achieve a balance between the legible and the pragmatic. When you type them in for yourself there are a number of changes you can make to condense them even further. These techniques are probably familiar to you and they include:

i) Omit spaces, except in PRINT statements (but watch it when you're editing). Howevernoticehow difficultitistoreadunspacedtext!

ii) Leave out the REMS.

iii) In the assembler programs, there is some pretty comprehensive annotation following the ';' signs. All of this can be omitted.

iv) Pack several statements on to one line and separate them with :'s. This is especially relevant to the assembler listings where each instruction is on a line of its own. There is no need for this – it just looks nice.

Well, now you should have grasped the central concept behind utility programming – so long as the program works, it should be condensed as much as possible. This method of coding will probably cause the utility to run faster, too. Basically what you are aiming for is a routine that works perfectly, occupies no memory and runs instantaneously! Well, there's no harm in trying!

Style

In many ways, the programming style which characterises these utilities has been determined by the requirements outlined above. Although these considerations place some restrictions on how the programs should be written, there is always plenty of scope for imposing one's own techniques and structures on the routines. We have allowed ourselves the full vocabulary of BASIC and have used whatever command we believed to be appropriate to the task at hand, regardless of theoretical considerations of parity. One thing you won't be seeing is THEN following IF; it is a noise word, so we don't use it – but there again, we never use LET, either!

Quite a number of routines are, of necessity, written in Assembly language. The BBC computer positively invites you to do this and, even if you have no experience of machine code, you should have no trouble getting these routines running. While this book is in no respect intended to be a machine code training manual, we hope that there is enough information in the program descriptions for the machine code novice to work his way through the Assembler routines. In some cases, the equivalent BASIC program is given to make you feel at home, and by comparing the two you should be able to follow the Assembler quite easily, assuming that you have some preliminary understanding of the language. Obviously the Assembler routines do not follow the same 'style' as the BASIC programs and neither are they documented in exactly the same way, although the really important details (what the program does and how you use it) are still there.

Most of the programs consist of a control section followed by a list of procedures, then finally any function definitions and DATA statements. We always use a single line (comprising line number, one blank space, then <RETURN>) to separate the various sections. These are never referred to and can be deleted if you wish.

As far as operation goes, there is very little that is common to all of the programs – which is hardly surprising since they are so varied. For example, some use the ESCAPE key as a valid input while others will just stop when you press it, so you will have to refer to the textual instructions for operating details. In fact the only common link between the programs, apart from the 'style' mentioned previously, is that where a key-press determines the program's progress, the ASCII code of the pressed key is stored in variable F% – the F stands for 'facility'.

The format of the documentation

Following its title, each utility is described under these headings:

Description:

This explains – in simple English – what the program does and why it might be used. In this section, no reference is made to the actual coding, only to what that coding achieves. For some programs, particularly the non-standard utilities, this section proposes a fairly detailed application of the routine.

Use

If a utility is not easy to use, then it is almost certainly a poorly conceived utility. By definition, these programs should save time and effort and if a result can be achieved more easily without the utility then it will never be used. For reasons already mentioned, none of these programs include instructions, and thus this section of the documentation will be the source of all operating details.

It is a sad fact of a utility's life that it often has to work under the most trying circumstances: in conjunction with other, more glamorous programs; stored at strange low addresses; packed on to function keys – it's no fun being a utility. This section suggests methods of operating the utility that are as convenient for you and the routine (in that order) as possible.

Most of the utilities in this book are obliged to work under unusual conditions and you should study this section carefully before you even attempt to load the routines.

How it works

This section describes, in very general terms, how the utility works. It does not go into any great detail about what variables are used or how a particular line works, but rather gives an overall picture of how the sections of the program are fitted together. Subsequent sections fill in the gaps of this brief description.

This method of describing programs – on a series of levels getting progressively more specialised – is appropriate to the style of the programs themselves, and the 'How it works' section explains the top level of this process. In a sense it is at this level where all the hard work comes, though it is precisely at this point where that becomes least apparent. If you think that the 'clever stuff' (such as it is) is in a few tricky lines of code at the end of the program, you have probably been beguiled by the apparent simplicity of the first section of the program. It is usually to these important early stages, where the flow of the program is mapped out, that this section refers. The fine detail of the program is described in the next two sections.

Procedures

The preceding paragraph refers to a style of programming in which structure plays an important part. The PROCedures are the building blocks that make up that structure. Most of the BASIC programs consist of a control routine followed by a list of PROCedures in a sensible order. The name of each PROCedure usually describes what it does, and this section expands that description. Theoretically, it should still be possible to state the purpose, if not the method of operation, of the PROCedure in general terms without referring to variables, though this is not a hard and fast rule.

Variables

Some variable names are retained throughout the length of a program; these are the major variables whose roles are always described. In addition, a program will contain a number of minor variables in different parts of the routine which are used to represent (slightly) different quantities. A good example of this would be the almost universal use of I, J and K as counts in various FOR. . .NEXT loops – no doubt a hangover from FORTRAN days. We are not keen on long variable names, so most are quite short and their use may not always be totally obvious – it is the purpose of this section to explain their function.

It costs nothing, in terms of memory, to use the resident integer variables A%-Z% and when one is used in an unobvious way, its use will always be explained.

Extensions

No program is ever finished – it is always possible to add bits here, remove a few lines there, to produce a whole series of variations on a single program. In this section we accept the fact that each program could do more (or less, depending on the application) and suggest some amendments that might be made. We have stopped at what seems to be a suitable point to provide a utility that can be put to good use right away. If you require a more specialised (or more general) routine then there should be enough information to help you write it, based on the utility that is already there. Also, it is in the spirit of this book that you do some work for yourself. As a programmer – why else would you be reading a book of utilities – you will be challenged, or goaded, into amending the programs and in this section we offer some advice on how to do so.

Some of these amendments are quite extensive (one may even be impossible) and some are trivial, but all are worth thinking about and you should benefit from so doing.

Saving the programs

We now consider the storage of our utilities. These are, after all, practical programs and it is important that they be located and loaded as quickly as possible.

Most professional or serious programmers will have a library of utilities – this can be a tape, or a disc devoted solely to utility programs and frequently-used procedures. You should start your own library if you have not already done so – why not begin with some of the routines in this book?

Utilities have been classified as either 'programs' or 'procedures' to be merged with your own programs, and these are best kept separate in your library. Also, your library will contain some of your most important programs, so it is a good idea to make regular backup copies. Throughout the book we assume that the code is held on some storage medium and then loaded into the computer where it remains undisturbed for as long as it is required. Details on how to do this are given with the individual routines. Let us briefly consider the most likely means of storage (in decreasing order of preference, all things being equal – they usually aren't so don't feel bad if you only have a cassette player, we'll look after you).

Tape

The obvious drawback is that retrieval from a cassette tape is very slow, although tape is cheap and readily available. Store the important routines at the beginning of the tape and have several tapes with the routines stored in different orders – one of them is bound to be near the program you require.

Learn to use the utilities sensibly. If you load up the Envelope Editor to design some envelopes for a game, do the lot at one go so that you only need to load the routine once. This kind of thing is fairly obvious and really comes down to good organisation on your part.

Disc

Without doubt, this is *the* place for utility programs. All disc owners should have a disc containing nothing but utility programs and, if you are fortunate enough to own a twin drive, this will probably be a permanent fixture in one drive, certainly during the development of a program. The Disc Filing System anticipates this and there is a command (`*LIB`) for setting up one drive/directory as a library from which programs can be `*RUN`. This is the optimum way of getting your programs to run (provided that they are written in machine code), and the library is where your utilities belong – it is designed for the job.

One small disadvantage of discs is that the DFS needs more space for itself than the cassette system and it creates this space by increasing the usual value of PAGE. Provided your DFS does not set PAGE beyond &1900 (this is most unlikely), you should have no problems with any of the programs in this book.

Notice, by the way, that the extra features available due to the disc system necessitate the inclusion of a whole host of new instructions to do the housekeeping. You don't need commands like *BUILD and *DUMP but they are useful time savers. In other words, they are utilities.

ROM

The idea of storing utilities on a chip is not new and tool-kit ROMS are available for quite a few computers. These extend the powers of the machine by providing a number of utilities – machine code monitors, printer drivers and the like. You may take the view that these chips only make up for what the manufacturer misses out, although that would be very unfair as far as the BBC Micro is concerned. Still, there is always room for more features and the very best place to have them is out of the way and yet on call on a ROM chip. This is about as close as you can get to the ideal utility described earlier – invisible, fast and readily available. Whether you are a tape or disc fan, you will have to admit that clearing a 'Bad program' by typing:

```
*BAD <RETURN>
```

is just about the ultimate in convenience!

A utilities ROM is a very practical possibility on the Beeb as the machine has been provided with some sideways ROM sockets. Admittedly, writing ROM-based utilities and then blowing them on to EPROMS is not everyone's cup of tea, but there are already several chips on the market to fulfill almost every conceivable need.

However, as far as this book is concerned, a ROM chip is not a practical place to hold your utilities and we shall stick to the rather more traditional storage mediums. Having had some experience with utility ROMS (both writing and using), let's point out one problem with even this seemingly idyllic situation. It is almost certain that whatever you are used to using, the ROM will have slightly different features that you would like to change. Tough. The whole point of ROMS is that you cannot (easily) change their contents and you are pretty well stuck with them. ROMS would not be a suitable medium for many of our programs because they are so open-ended, and may be changed to suit the requirements of the user (I had to put this in so that tape and disc users don't get too jealous – actually, I have a utility ROM and it's extremely useful).

The operating instructions supplied with each routine assume that you have the program on either tape or disc, and any problems in this respect will be explained.

IMPORTANT NOTE

The BBC computer comes in a variety of guises and not all the programs in this book can be guaranteed to work on all systems. The utilities were written on a BBC Model B (32K RAM), fitted with Operating System 1.2, BASIC 1 and an Acorn DFS, and will obviously work on such a system (or similar but without the DFS).

NO ATTEMPT HAS BEEN MADE TO MAKE THESE PROGRAMS COMPATIBLE WITH O.S.0.1. All new computers sold should be fitted with O.S. 1.2 and nearly all owners will have upgraded their machines to this version. It is quite likely that future releases from most software houses will expect to run on machines fitted with this O.S., and if you still have the 0.1 version, you are strongly urged to upgrade it as soon as possible.

Many computers are now fitted with BASIC 2 and some slight modifications will be needed to programs that use the ',' delimiter in INPUT statements – this should now be changed to ';' as used in other BASICS. There are other differences between the two BASICS but none of these should affect our programs.

If your computer has 16K RAM, then obviously very long programs, or those that use MODES 0–3, will not run on your machine without some modifications. We have not tried to indicate which programs will work on the 16K machine – just get that extra 16K!

In theory, any program that works with a disc machine will work on a tape machine as well (unless it is on disc, of course). However, many of the machine code routines use a 'safe' area for the assembled code, namely Pages 9 and 10 (&900–&AFF), but this area is not so safe when the computer uses the cassette system. The code is assembled here because it should be in a fixed place so that later routines know where to look for it, and these Pages are (usually) as safe as anywhere. However, if you use ENVELOPE numbers greater than 4, the definitions overflow into Page 9 and will probably (you may be lucky) corrupt anything in there.

With the cassette system this space is used for output and input by:

Page 9: *SPOOL, BPUT
Page10: *EXEC, BGET

and these commands are guaranteed to overwrite anything in those Pages. Once you are aware of this, it should not be too difficult to plan round the problem. If you store anything in, say, Page 9 you can reasonably be expected to find it there later on when you need it. Be warned: some of our utilities use BPUT and BGET – or you might use them, or *EXEC and *SPOOL, yourself – so try to anticipate this and use the routine accordingly.

This is not a serious problem, but it should be mentioned here. In most cases any code that is stored in this area is only needed for as long as a particular routine is being used. Once the calling routine is removed, the machine code may as well be too.

Section 1:

Basic utilities

In this section we look at some routines which all programmers should have in their library of utilities. These routines form the basis for any diagnostic or developmental work on a micro – for example, many of the programs in this book were written following an initial investigation with the memory display utility.

Because of their work-a-day nature, these programs offer few gimmicks and have no facilities beyond their intended purpose. Explanatory messages and error trapping have been kept to a minimum so that the programs are as short (and therefore as versatile) as possible. Consequently, if you want to get the most out of each utility you should read the instructions relating to each routine very carefully.

Each of the utilities in this chapter expects input, and delivers output, in hexadecimal notation which comes rather more naturally to computers than it does to humans. Mathematically speaking, this means that numbers are represented in base 16 with the labels A to F replacing the 'digits' 10 to 15. Hexadecimal ('hex' for short) is now universally used to represent addresses and data within computers, and an understanding of hex is essential for the correct use of the programs in this chapter (and for serious programming generally).

On the BBC machine, numbers prefixed by '&' are taken to be hex whereas numbers prefixed '~' (tilde) will be converted to hex from decimal for, say, printing purposes. Furthermore, facilities exist for manipulating hex data, notably the '?' (query) and '!' (pling) indirection operators. Other nice features of interest to the keen programmer include a variation on the DIM statement which allows you to construct byte-sized tables, and a string indirection operator, $.

If you are not familiar with these concepts then you should refresh your memory from the *User Guide*.

Utility 1:

Scrolling memory editor

Description

In many respects this, or something like it, is the single most important utility a programmer can have and we give three different versions of the utility, each having its own particular use.

The routines display, in hex, the contents of a chosen section of memory, either ROM or RAM, complete with the addresses, also in hex. Such a display is often called a 'memory dump'. If the contents of any location lie within the ASCII range 32–126, then the character corresponding to that byte is also printed. This makes it easy to locate items (e.g. programs) in store.

The first version of this routine is the deluxe model (i.e. it includes colour!), which allows forward and backwards scrolling and includes an edit facility. Using a MODE 7 screen, each display line consists of an address (in red), the contents of eight bytes of memory and, if appropriate, their (green) character representations. To make the listing easy to read, those bytes within the ASCII range (32–26, or &20 – 7E in hex) are printed in cyan, while other characters are printed in magenta.

In the edit mode you can easily change the contents of an area of RAM by entering hex bytes. This is a really useful facility which allows you a great deal of control over the computer. However, you should be careful where you write data if you are not familiar with the memory map of the machine. Obviously you cannot write to ROM, but no damage will result if you attempt to do so.

Use

Run the utility and enter the hex address from which the dump is to start. Do not prefix the address with a '&' character – the program does it for you. The following keys are effective:

A scrolls the display UP by adding a new line at the bottom of the screen.

B scrolls the display DOWN by adding a new line at the top of the screen. Both the **A** and **B** options auto-repeat and, holding the keys down, will generate about four lines per second.

E enters 'edit mode'. The top line of the display can be edited by typing in single bytes (in hex) followed by <RETURN>. The indicator (V) shows the byte to be written and it is automatically stepped on to the next byte, with scrolling if necessary. If you do not want to amend a byte, pressing <RETURN> will skip past it and indicate the next one.

If, during editing, you cause an error by entering non-hex data or by pressing ESCAPE you will be returned to the scrolling mode.

ESCAPE will quit the program from scrolling mode.

```
 10 REM SCROLLING MEMORY EDITOR
 20 MODE 7
 30 *FX 12,1
 40 S$="12345678"
 50 INPUT "Start address (hex) "A$
 60 addr%=EVAL("&"+A$):tlhc%=addr%
 70 FOR I%=1 TO 23
 80 PROCline(addr%):addr%=addr%+8
 90 NEXT
100
110 ON ERROR GOTO 650
120 REPEAT
130 F$=INKEY$(0)
140 IF F$="A" PROCup
150 IF F$="S" PROCdown
160 IF F$="E" PROCedit
170 UNTIL FALSE
180
190 DEFPROCup
200 VDU 28,0,24,39,1,31,0,22,10
210 tlhc%=tlhc%+8
220 PROCline(tlhc%+176)
230 ENDPROC
240
250 DEFPROCdown
260 VDU 28,0,24,39,1,30,11
270 tlhc%=tlhc%-8
280 PROCline(tlhc%)
290 ENDPROC
300
310 DEFPROCedit
320 ON ERROR VDU 7:GOTO 110
330 REPEAT
340 addr%=tlhc%
350 FOR disp%=0 TO 7
360 VDU 26,30
370 PRINT SPC(6+3*disp%)"V"CHR$152
380 VDU 23,1,1;0;0;0;
390 INPUT TAB(1,0)":"A$
```

```
400 IF A$<>"" PROCupdate
410 NEXT
420 PROCup
430 UNTIL FALSE
440
450 DEFPROCupdate
460 addr%?disp%=EVAL("&"+A$) AND &FF
470 PROCline(tlhc%)
480 ENDPROC
490
500 DEFPROCline(X%)
510 X%=X% AND &FFFF
520 VDU 23,1,0;0;0;0;129
530 @%=4:PRINT ~X%;
540 @%=1:S$=CHR$130
550 FOR J%=0 TO 7
560 byte%=X%?J%:col%=134:A$=CHR$byte%
570 IF (byte%<32 OR byte%>126) col%=133:A$=" "
580 VDU col%,-48*(byte%<16): PRINT ~byte%;
590 S$=S$+A$
600 NEXT
610 PRINT S$
620 *FX 15,1
630 ENDPROC
640
650 *FX 12,0
660 VDU 26,31,0,24,10,23,1,1;0;0;0;
```

How it works

Having accepted your starting address, the program enters a loop, waiting for you to press one of the relevant keys. The loop is endless, and can only be left with ESCAPE.

To move the display up the screen, the cursor is moved to the bottom line and then a 'cursor down' command is sent, thus forcing the display up by one line. Similarly, to scroll the display down, a 'cursor home' followed by a 'cursor up' is used. This forces the display down so that the new line can be slotted into place.

In the edit mode, input bytes are EVALuated and written to memory, if they are acceptable (that is, they must be valid hex and the destination address has to be somewhere in RAM). Each time a byte is amended the top line is rewritten so that both hex and ASCII displays are updated.

Procedures

Pressing A, S or E causes the appropriate PROCedure to be called, these being PROCup, PROCdown and PROCedit respectively and the purpose of each should be clear from the preceding description. PROCupdate is called by PROCedit to store the new byte just entered and to update the top line of the display by calling PROCline. This is the main PROCedure and is responsible for printing an entire line consisting of address, eight hex bytes and ASCII data to the screen. The parameter X%, which is passed to the PROCedure, is the address of the first byte on the line. PROCline does not update any variables, it only prints out data.

Variables

In this particular program, most variables are 'local' to a small section of code and so the same names can be used in different areas without causing any problems. For example, A$ is used throughout as a general purpose string and addr% as a general purpose address. Although considered to be bad practise, this minimises memory usage, which is a prime consideration with some of these routines.

The entire display is determined by the address of the byte in the top left-hand corner and so this value is held in tlhc%, which is the really important variable in this program. tlhc% must be updated each time the display scrolls up or down.

Other significant variables are byte%, which contains the contents of the location currently under scrutiny, and S$ which is the ASCII string built up as the line is processed.

Both this and the following programs use the PRINT formatting variable @% to produce a neat display. Addresses are printed with @%=4 so that they are right aligned in a four byte field. When hex data is to be PRINTed, @% is set to 1 so that the program, rather than the computer, is responsible for PRINTing out leading and trailing spaces.

Notice also the use of the tilde (~) facility to PRINT out bytes in hex format.

Extensions

The next two programs are just two of many possible variations on this routine. They are presented as separate programs so that you may choose the right utility for the job without including features (and thereby wasting potentially valuable memory) that will not be required. As far as this routine is concerned, one improvement would be to include the extensive editing facilities used in the Disc Sector Editor utility that appears later. This would enable you to edit both hex and ASCII in a more versatile way and may be worth doing if you intend to make much use of that particular option.

Utility 2:

Single key memory display

Description

It is possible to condense the essential features of the previous routine so that a dump can be made available simply by pressing a function key. This utility will provide you with a scrolled memory dump, starting at your chosen location, until you press ESCAPE. It has no frills whatsoever and is entered in minimal BASIC abbreviations so that it occupies the minimum amount of space in the function key buffer. Because this coding is so difficult to follow, the listing includes an expanded version so that you can see how the *KEY9 version works.

Use

It is only necessary to enter the text of line 30 (i.e. without the line number) to set up key f9 to do the dump. However, even that is tedious, and you might like to save a one-line BASIC program consisting of line 30 on its own. CHAINing this would then set up the key for you.

To use the routine, press f9 and enter the starting address in hex when the '?' prompt appears. Use SHIFT to provide another page of the dump, and ESCAPE to quit the program.

```
   10 REM SINGLE KEY MEMORY DISPLAY
   20
   30 *KEY9 ¦NI.A$:A=EV.("&"+A$):REP.:A$
="":@%=4:P.~A"   ";:@%=1:F.I=0TO7:B=A?I:V
.-48*(B<16):P.~B" ";:A$=A$+CHR$(B*(B>32)
*(B<127)-32*((B<33)+(B>126))):N.:P.A$:A=
A+8:U.FA.¦M
   40 END
   50
   60 REM EXPANDED VERSION
   70 VDU 14
   80 INPUT A$
   90 A=EVAL("&"+A$)
  100 REPEAT
  110 A$=""
  120 @%=4
  130 PRINT ~A;"   ";
  140 @%=1
  150 FOR I=0 TO 7
  160 B=A?I
  170 VDU -48*(B<16)
  180 PRINT ~B;" ";
  190 A$=A$+CHR$(B*(B>32)*(B<127)-32*((B
<33)+(B>126)))
  200 NEXT
  210 PRINT A$
  220 A=A+8
  230 UNTIL FALSE
```

How it works

There is nothing difficult about the expanded version but notice the use of relational operators (ROs) in lines 170 and 190. If you are not familiar with ROs then you soon will be as they are used quite a lot in this book, though only where appropriate. The *User Guide* touches on them in a section entitled 'More on TRUE and FALSE' but does not quite give the full story. Here is a brief explanation of line 170:

The computer compares B with 16 and, if it is less, the expression (B<16) is given the value -1, or TRUE, otherwise (B<16) is given the value 0, or FALSE. In other words, when the computer sees B<16 it asks itself 'IS B<16?' and answers accordingly. If you think about it, you will realise that line 170 is really a shorter way of saying:

IF (B<16) VDU 48 ELSE VDU 0

or, in view of what VDU 0 does:

IF B<16 VDU 48

However, we must remember that this statement is part of a single line and that, should the IF fail, BASIC goes looking for the next line – naturally it won't find it as there isn't one and so the program stops. By using the relational operators we avoid this problem.

Incidentally, the purpose of this line is to prefix hex items that would only consist of a single byte with a leading '0', thereby ensuring a nice tidy display.

A more complicated example appears in line 190 and its purpose is to PRINT CHR$(B) if B is in the range 32 < B< 127 and otherwise to PRINT CHR$(32) – i.e. a space. See if you can convince yourself that it will do this.

Extensions

Considering the size of the function key area, it is unlikely that this routine will expand much further. It may just be possible to include the rather nice scrolling effect of the previous program, although we suspect that it cannot be done.

You can consider this to be a challenge.

Utility 3:

Printer memory dump

Description

There is really little to say about this except that it is a minor modification of the previous routines to enable the memory dump to be sent to a printer. The printer should be capable of outputting at least 74 columns as each line contains the data for 16 bytes instead of the 8 used by the screen routines.

A sample of the output produced by this utility is shown in Fig. 1 in the next section.

Use

Where you LOAD the program depends on the other occupants of the machine at the time – obviously you do not want to LOAD it in on top of the very data you are trying to print out! Because the routine is so short, it will fit into Pages 9 and 10 where it is out of harm's way, so before you LOAD it off tape or disc, set PAGE=&900 and then reset it to its normal value when the LOADing is complete. To summon up the routine, you only need to enter:

PAGE=&900 <RETURN> then RUN <RETURN>

Alternatively, set up a function key to do the job for you.

Once again you must enter the start address in hex but, this time, exit by pressing 'X'. To keep the printout tidy the current line is completed before printing stops.

```
   10 REM PRINTER MEMORY DUMP
   20 S$=STRING$(16," ")
   30 INPUT "Start address (hex) "A$
   40 A%=EVAL("&"+A$)
   50 VDU 2,15
   60
   70 REPEAT
   80 @%=4
   90 PRINT ~A% AND &FFFF;" ";
  100 @%=1:S$="  "
  110 FOR J%=0 TO 15
  120 B%=A%?J%
  130 IF (B%<32 OR B%>126) A$=" " ELSE A
$=CHR$B%
```

```
140 VDU 32,C%,-48*(B%<16)
150 PRINT ~B%;
160 S$=S$+A$
170 NEXT
180 PRINT S$
190 A%=A%+16
200 UNTIL INKEY$(0)="X"
210 VDU 3
```

Extensions

Here is another candidate for compressing on to a function key: by referring to the previous routine you should have no difficulty in doing so if it is really necessary, although there is little point in moving it from Page 9 unless you need to store something else there.

Because this program has so much in common with the others it could be combined with them to produce a more comprehensive (but necessarily longer) routine. Whether you do this will depend on your intended application for the program.

Despite their simplicity, these three programs are extremely useful diagnostic aids and should be near the beginning of your utilities tape/disc. They tell you (nearly) everything that is going on inside the computer and, provided you know that to look for, they will help you to answer 'What went wrong?' in most cases.

Their use is not restricted to fault-finding, however, and they can reveal many interesting facts about the machine. We shall refer back to these memory display utilities at various points throughout the book.

Utility 4:

Disassembler

Description

One of the best features of the BBC machine is the built-in Assembler. This makes it possible to write (source) code using 6502 mnemonics and symbolic labels which are then translated into machine (object) code by the Assembler. A disassembler performs the reverse process and is absolutely essential for anyone writing – or even merely interested in – machine code.

The output from this disassembler is similar to that of the Assembler except that:

i) labels are not given

ii) ASCII equivalents are supplied

iii) branch instructions are provided with a direction and the absolute destination address

iv) we use colour

During disassembly, any invalid op-codes are assumed to be single byte instructions and are replaced by '???'.

As it can be difficult to follow disassembled code on the screen, you are given the option of sending output to the printer when you first run the utility.

At this point you may feel inclined to turn to another section on the grounds that you are not yet *au fait* with machine code. If not, why not?? You have the ideal machine on which to learn and there are now numerous books and articles on the subject. The BBC computer has been well designed to make machine code programming as painless as possible and it is well worth making an effort to learn. Throughout the book I assume that you are prepared to have a go at understanding the assembler routines even though you may prefer to be reading BASIC.

If you are still somewhat apprehensive about machine code a good place to start is to type in this disassembler and use it to look at how other people write programs. There are many machine code programs available for the BBC Micro and you can learn a great deal just by studying them with the disassembler. In addition, several of our utilities are written in machine code, although you do not need to be familiar with machine code to get them to work or to understand the principles underlying their operation.

Use

Normally it is best to LOAD the disassembler at the usual value of PAGE and then set PAGE=PAGE+&C00 so that programs may be LOADed and run without affecting it. To use the utility, reset PAGE to the correct value and then RUN. Select 'N' in response to the question 'Disassemble to printer?', and enter the address (in hex) at which disassembly is to commence. For starters, try an address somewhere in BASIC, i.e. between &8000 and &BFFF – you will then see how the experts write machine code. If the result of this is a whole mass of ???'s (invalid op-codes) it is because you have landed in the middle of a data table rather than executable code. BASIC contains several such tables, so try a different address if you find one.

Pressing key 'A' will generate one line of output and if you hold it down lines are produced at the rate of about three per second. If you press ESCAPE, you are returned to the question 'Start address?' so that you may continue disassembly from a different point. To exit the program, you should press ESCAPE in response to this question.

This approach will not be so successful if the program you want to disassemble has to be LOADed in at PAGE for correct operation (as might be the case for, say, a video game). Any absolute addresses within the program would be displaced by an amount: (actual LOAD address - true LOAD address), making the code very difficult to follow. The simple solution to this is to LOAD the disassembler in a different place – for example, near the top of memory to make room for the intended disassemblee (if I may coin a word). Before doing so you should switch to MODE 7 and reserve at least 12 pages (&C00 bytes) for the routine.

Alternatively, the disassembler can be modified to include an offset facility so that it can stay where it is and pretend that the code it is disassembling is actually located somewhere else. We will look at the offset facility in the Extensions section below.

```
   10 REM DISASSEMBLER
   20 MODE 7 : @%=1
   30 INPUT"Disassemble to printer (Y/N)
 ",A$
   40 IF A$="Y" PROCprint ELSE PROCscree
n
   50 ON ERROR GOTO 960
   60 VDU 3,28,0,1,39,0,12
   70 INPUT "Start address: &"A$
   80 pc%=EVAL("&"+A$):start%=pc%
   90 ON ERROR GOTO 990
  100 VDU 28,0,24,39,2,12,vdu%
  110 REPEAT
  120 IF vdu%=2 OR INKEY(0)=65 PROCline
  130 UNTIL FALSE
  140
  150 REM ONE S/R PER ADDRESSING MODE
  160
  170 RETURN : REM SOME SUBROUTINE!!!
  180 PRINT "A";:RETURN
  190 PRINT "#";pc%?1;:RETURN
  200 GOSUB 220:PRINT ",X";:RETURN
  210 GOSUB 220:PRINT ",Y";:RETURN
  220 PRINT "&";:PROChex(pc%?1):RETURN
  230 d%=pc%?1:to%=pc%+2+d%:X$="+"
  240 IF d%>127 to%=to%-256:X$="-":d%=25
6-d%
  250 PRINT X$;d%;C1$;"(";
  260 GOSUB 330:PRINT ")";:RETURN
  270 PRINT "(&";:PROChex(pc%?1):PRINT "
,X)";:RETURN
  280 PRINT "(&";:PROChex(pc%?1):PRINT "
),Y";:RETURN
  290 GOSUB 320:PRINT ",X";:RETURN
  300 GOSUB 320:PRINT ",Y";:RETURN
  310 PRINT "(";:GOSUB 320:PRINT ")";:RE
TURN
  320 to%=(pc%?1)+256*(pc%?2)
  330 PRINT "&";:PROChex(to% DIV 256):PR
OChex(to% MOD 256):RETURN
  340
  350 DEFPROCprint
  360 C1$=" ":C2$=" ":C3$=" ":C4$=" ":C5
$=" "
  370 vdu%=2:ENDPROC
  380
  390 DEFPROCscreen
  400 C1$=CHR$129:C2$=CHR$130:C3$=CHR$13
1:C4$=CHR$133:C5$=CHR$134
  410 vdu%=15:ENDPROC
  420
  430 DEFPROCline
  440 pc%=pc% AND &FFFF
  450 PRINT C2$;
  460 PROChex(pc% DIV 256)
  470 PROChex(pc% MOD 256)
  480 PRINT C3$;
  490 byte%=?pc%
  500 IF (byte% AND 3)=3 byte%=3
  510 RESTORE
  520 FOR I%=0 TO byte%-(byte% DIV 4):RE
AD code$:NEXT
  530 am%=ASC(code$)-96
  540 mn$=RIGHT$(code$,3)
  550 asc$=""
  560 ex%=-(am%>2)-(am%>9)
  570 FOR I%=0 TO ex%
  580 asc%=pc%?I%
  590 PRINT C3$;
  600 PROChex(asc%)
  610 IF asc%<32 OR asc%>126 asc%=32
  620 asc$=asc$+CHR$(asc%)
  630 NEXT
  640 REPEAT:PRINT " ";:UNTIL POS=16
  650 PRINT C4$;mn$;"   ";
  660 ON am% GOSUB 170,180,190,220,200,2
10,230,270,280,320,290,300,310
  670 REPEAT:PRINT " ";:UNTIL POS=35
  680 PRINT C5$;asc$
  690 pc%=pc%+ex%+1
  700 *FX 15,1
  710 ENDPROC
  720
  730 DEFPROChex(X%)
  740 VDU -48*(X%<16):PRINT ~X%;
  750 ENDPROC
  760
  770 DATA aBRK,hORA,a???,a???,dORA,dASL
,aPHP,cORA,bASL,a???,jORA,jASL
  780 DATA gBPL,iORA,a???,a???,eORA,eASL
,aCLC,lORA,a???,a???,kORA,kASL
  790 DATA jJSR,hAND,a???,dBIT,dAND,dROL
,aPLP,cAND,bROL,jBIT,jAND,jROL
  800 DATA gBMI,iAND,a???,a???,eAND,eROL
,aSEC,lAND,a???,a???,kAND,kROL
  810 DATA aRTI,hEOR,a???,a???,dEOR,dLSR
,aPHA,cEOR,bLSR,jJMP,jEOR,jLSR
  820 DATA gBVC,iEOR,a???,a???,eEOR,eLSR
,aCLI,lEOR,a???,a???,kEOR,kLSR
```

```
  830 DATA aRTS,hADC,a???,a???,dADC,dROR
,aPLA,cADC,bROR,mJMP,jADC,jROR
  840 DATA gBVS,iADC,a???,a???,eADC,a???
,aSEI,lADC,a???,a???,kADC,a???
  850 DATA a???,hSTA,a???,dSTY,dSTA,dSTX
,aDEY,a???,aTXA,jSTY,jSTA,jSTX
  860 DATA gBCC,iSTA,a???,eSTY,eSTA,fSTX
,aTYA,lSTA,aTXS,a???,kSTA,a???
  870 DATA cLDY,hLDA,cLDX,dLDY,dLDA,dLDX
,aTAY,cLDA,aTAX,jLDY,jLDA,jLDX
  880 DATA gBCS,iLDA,a???,eLDY,eLDA,fLDX
,aCLV,lLDA,aTSX,kLDY,kLDA,lLDX
  890 DATA cCPY,hCMP,a???,dCPY,dCMP,dDEC
,aINY,cCMP,aDEX,jCPY,jCMP,jDEC
  900 DATA gBNE,iCMP,a???,a???,eCMP,eDEC
,aCLD,lCMP,a???,a???,kCMP,kDEC
  910 DATA cCPX,hSBC,a???,dCPX,dSBC,dINC
,aINX,cSBC,aNOP,jCPX,jSBC,jINC
  920 DATA gBEQ,iSBC,a???,a???,eSBC,eINC
,aSED,lSBC,a???,a???,kSBC,kINC
  930
  940 REM ERROR HANDLING
  950
  960 IF ERR=28 GOTO 60
  970 IF ERR<>17 GOTO 1000
  980 VDU 12,26:END
  990 IF ERR=17 GOTO 50
 1000 PRINT '"Error at ";ERL
 1010 REPORT:PRINT:END
```

How it works

Writing a disassembler is really an exercise in data organisation, and any such program is bound to include one or more look-up tables to decode data into the various formats required.

Given a starting address, the program assumes that a valid 6502 instruction will be found there; this enables it to work out how many of the following bytes belong to that instruction and consequently where the next instruction starts. The whole sequence is then repeated so that disassembly carries on *ad infinitum* by a sort of inductive process. Theoretically, if you start in the wrong place – that is, the middle of an instruction – all the following code will be out of synch. In practice, it usually falls into line within very few instructions.

Using a 256-entry table (actually not strictly true, but let's think of it like that for now) the first byte is converted to:

i) An addressing mode a–m (or 1–13)

ii) A mnemonic op-code ADC – TYA and including ???

The addressing modes are arranged so that the number of the mode indicates the expected length of the instruction and we don't need an extra table to work that out. The addressing mode is read into variable am% and the total length of the instruction is computed as follows:

am% range	number of bytes
am%<3	1
2<am%<10	2
am%>9	3

Using relational operators (remember them?), this can be condensed to a single line (line 560). Once the number of bytes is known, we can print out the hex part of the display with, since we looked it up first, the mnemonic op-code. Now, by consulting am% again we can decide the format of the rest of the instruction by calling one of thirteen subroutines.

Did I really say subroutine? I'm afraid so – for the first and only time in this book we have to use the old-fashioned GOSUB command because of its compatibility with ON. Each subroutine prints out the operands following the op-code even if, as in the case of 'implied' instructions, that means printing nothing at all. Because the ON statement (line 660) is so long, the subroutines are located near the beginning of the program so that they have three-digit line numbers. This does not affect the length of line 660 as stored, but only when it is printed out. Normally, of course, the subroutines could be expected to be found towards the end of the program. The relationship between am% and the addressing modes is shown in Table 1.

Finally, the ASCII data is printed out and all pointers updated ready to start on the next instruction.

The *User Guide* contains a list of all 6502 op-codes arranged alphabetically. If you rearrange this table into numerical order, starting with 0 for BRK, you will see several patterns in the way the codes are allocated. The most striking one, which we take advantage of here, is that any op-code of the form &x3, &xB, &xF, where x is any hex digit, is invalid. To the mathematician these bytes would all be 'congruent to 3 and 4', or to the lay-person, when you divide one by 4 the remainder is always 3. There are other invalid op-codes too, but this simple observation means that we can eliminate a quarter of our main look-up table. Of course, we cannot use the same look-up criteria as if the table had 256 entries, but the amendment (see lines 500, 520) is hardly extensive.

Each entry in the table – that huge block of DATA statements – consists of the addressing mode, in lower-case for clarity, and the actual mnemonic, which is always three characters, required by that byte. From this information, the format of the line can be pieced together.

It has already been mentioned that output can be sent to the printer. To do so, answer 'Y' to the first question and then supply a start address. When the printer option is selected the screen prints in white and the printing is automatic – no need to press the 'A' key. ESCAPE is used in the same way as before.

am%	Address Mode
1	Implied
2	Accumulator
3	Immediate
4	Zero-page
5	Zero-page,X
6	Zero-page,Y
7	Relative (branch)
8	(Indirect,X)
9	(Indirect),Y
10	Absolute
11	Absolute,X
12	Absolute,Y
13	Indirect

Table 1. The internal (am%) code used by the disassembler for each 6502 addressing mode.

Procedures

When outputting data to the screen it is nice to include a dash of colour, although a printer might get confused by any colour control codes. To avoid this, constants C1$ - C5$ are set up to contain spaces or colour codes by PROCprint and PROCscreen respectively.

The main procedure of this program is PROCline which is responsible for printing one whole line of the display; each line consisting of one disassembled instruction. When PROCline is called, variable pc% points to the first byte of the instruction and on exit, pc% will be incremented by the length of this instruction and, as a result, will point to the next one.

PROChex is a simple one-liner to print out the parameter X% as two hex digits.

Variables

Throughout the program, pc% is used to point to the instruction currently being disassembled. It is an address and stands for 'program counter' which, in machine language, is the name given to the address of the instruction being obeyed by the microprocessor (or CPU – its use is not restricted to micros). The contents of location pc% are called byte%. Initially, byte% is converted to code$ which is one of the entries in the data table and this is further subdivided into am%, the addressing mode referred to earlier, and to mn$, the 6502 mnemonic op-code.

One very important variable is ex%, which is the number of bytes expected by the instruction. It can take the values 1, 2 or 3 and is needed to update pc% so that it can point to the next op-code. While the instruction is being decoded, asc$ is built up ready to be output as the last part of the line.

Extensions

The number of extra facilities that can be included in this disassembler really depends on your programming requirements. A simple amendment would be to print out operating system calls by their vector names, rather than in hex. Thus:

```
JSR &FFE3
```

would become

```
JSR OSASCI
```

which is rather easier to follow. To include this facility would mean adding a new table to the program – you would have to decide if the benefit of the facility would out-weigh the loss of memory involved.

A really nice touch would be to include a 'backward' scroll key as used in the memory editor. It is the nature of 6502 machine code that it can only safely be read forwards since the instructions are of varying length. This does not mean that it is impossible to work backwards, only that any attempt to do so would have to involve an educated guess and the best you could hope for would be to optimise that guess. To understand the problem consider this example:

```
E82  C6       ?
E83  AC       ?
E84  A5       ?
E85  48       ?
--------------------------------- we are here
E86  20 EE FF  JSR &FFEE
```

Suppose that we are at the dotted line and we know that this really is a *bona fide* instruction. What do those preceding bytes mean? They could mean: DEC &AC:LDA &48 if we were to start reading them from &E82. However, they could equally well stand for: (&C6):LDY &48A5, where the C6 belongs to some earlier instruction. On the other hand they may just be data and mean nothing at all! Clearly you would have to make the best of this and one possible algorithm to help you decide is to go back say, 12 bytes, and assume that you are at the start of an instruction. Disassembling forward to the present location may work perfectly or it may introduce a few unrecognised op-codes. If so, try again, this time going back by 11 bytes and, if necessary, a final attempt to going back by 10 bytes (it is pointless trying more than three times as the longest instruction occupies 3 bytes). Whichever of these gives the best result may be taken as the correct one.

This is extremely imprecise, as it is bound to be, and it is probably more of an academic exercise (although a good one) than a practical addition to the routine. I must admit though, it would be very nice to scroll through pages of disassembled code in either direction safe in the knowledge that it was all correct. Maybe one day. . .

Earlier we hinted at the idea of introducing an 'offset' facility into the disassembler and this is a relatively straightforward amendment. Having just bought Super-Pac-Alien-Nightmare-Kong (the latest video game – a big hit in its abbreviated form) you will no doubt be anxious to disassemble it to see how it works. Unfortunately, being in machine code, it may well LOAD up at some horribly low address (eg. &900) where you would prefer not to LOAD it.

If you force it to LOAD at, for example, &2000 and disassemble it from there, any absolute addresses within the program will be out by &2000-&900=&1700. Using the new version of the disassembler you would reply 1700 in response to the question 'Offset' and the routine would then adjust all the addresses to suit.

Addresses that would normally lie between the start of the program and &7FFF are the only ones affected – all operating system calls, etc., are left alone. You will have an indication of what has been changed since the hex bytes will be printed 'honestly', in all cases.

To add the offset facility to the program these lines have to be changed:

```
  43 INPUT '"Offset ",A$
  46 off%=EVAL("&"+A$)
 235 to%=to%+off%
 325 IF to%>start% AND to%<&7FFF to%=to
%+off%
 460 PROChex((pc%+off%) DIV 256)
 470 PROChex((pc%+off%) MOD 256)
```

Clearly this version of the disassembler with an offset of 0 is equivalent to the original version. You might like to stick with the new version as it is more versatile and only very slightly longer. Personally, I found myself using an offset of 0 so often that I found the facility to be rather redundant.

Finally, we look at another variation on the disassembler theme; this is the one which allows you to follow branches and jumps.

When you are trying to follow a section of code on the screen, you will often come across branches, JSRs and the like to code off the screen and you may wonder what is going on there. Having got there, you may wish you'd never been (like a holiday in Southport – sorry Southport, don't mean it) and that you were back on the original path. Using this variation, each time you come across a branch – JMP, JSR, or (provided you have JSR'ed at some point) an RTS or RTI – you have the option of following that path or of continuing disassembly as though nothing had happened.

As it turns out, this is not too difficult; the amendments you need are:

```
   25 DIM stack%(20)
   85 sp%=0
  245 branch%=2
  445 branch%=0
  545 IF (byte%=&60 OR byte%=&40 AND sp%
>0) OR ASCmn$=74 branch%=1
  691 IF branch%=0 GOTO 700 ELSE VDU 7
  692 REPEAT UNTIL INKEY$(-66)=0
  693 *FX 15,1
  694 IF GET<>32 GOTO 440 ELSE PRINT
  695 IF byte%=&4C OR branch%=2 pc%=to%
  696 IF byte%=&6C pc%=?to%+256*(to%?1)
  697 IF byte%=&20 sp%=sp%+1:stack%(sp%)
=pc%:pc%=to%
  698 IF byte%=&60 or byte%=&40 pc%=stac
k%(sp%):sp%=sp%-1
  699 GOTO 440
```

This version occupies more memory than the others and so you should set PAGE=PAGE+&E00 before running the program. Everything will work as before except that the display will stop and the computer will bleep when it comes to a branching instruction. To ignore the branch press 'A' as usual (you will have to release 'A' first if it is held down) or, to follow the new path, press the space bar. If you choose the second option the next line is printed following a blank line to indicate a change in direction. Everything now functions as before with the 'A' key providing you with one (or several, if held down) lines of disassembled code.

If the computer bleeps on an RTS instruction, you must have previously followed a JSR and pressing the space bar will return you to the instruction immediately after the JSR in the traditional way.

To implement this facility we have to have a stack – not surprisingly called `stack%(20)` – and a stack pointer which appears in the amendment as `sp%`. These are updated at the appropriate times (have a look at the listings to see if you can work out when they occur) and will allow you to follow a chain of subroutines down to a depth of 20.

At the start of a line, `branch%` is initialised to 0 and it is then set if the instruction turns out to involve some form of branch. If `branch%` is set, the nature of the branch is analysed in lines 694–698 and the appropriate action taken.

Utility 5:

Block move

Description

This utility will move a section of memory – for example, a program – to a new location anywhere in RAM. The only memory affected by the move is the receiving area and the new location may overlap the old one without any corruption of data.

Two possible uses for this routine are: i) moving long disc-based programs down to &E00 so that they will run and ii) creating 'cloned' copies of programs in RAM so that one may be used for experimental purposes. This is not as crazy as it sounds. Suppose you have this utility in store at PAGE and you want to move a program down to overlay it. Clearly, you cannot use the BLOCK MOVE routine to do the move as it is about to be obliterated – if this happens you may end up with two corrupted programs on your hands; both the utility and the program you were trying to move. One way out would be to use the utility to reproduce itself (they can't touch you for it) somewhere else in memory and then use *that* version for the move.

Use

As we have just seen, the initial position of the routine in store is not too important as it can always move itself off somewhere else. However, because it is a BASIC program it should always be located at a Page boundary.

When RUN the routine requests three addresses, all in hex (but you mustn't enter the '&') – they are, respectively, the first and last bytes of the block to be moved and the new position of the first byte.

If you want to see the routine in action, type MODE 7 <RETURN> and then RUN<RETURN>. Enter the addresses 7C00, 7FFF and 7C03.

The contents of the screen will slide three bytes to the right, starting with the bytes at the bottom right of the screen and ending with the byte in the top left-hand corner. This is because &7C00 is the start of the screen memory in MODE 7 and this little exercise will give you some indication of the speed of the move.

Another interesting experiment is to now LIST the program and re-RUN (without typing MODE 7) using the same addresses. The overall effect will be the same but notice that the movement starts and finishes in the middle of the screen, which seems to suggest that the screen RAM doesn't start at &7C00 any more! The explanation of this effect is that the BBC machine uses a technique known as 'hardware scrolling' which basically means that it reserves the right to decide where the screen RAM starts, within the range assigned to the screen. This enables the computer to perform extremely fast listings as it hardly needs to reorganise the lines on the screen whilst it is scrolling them.

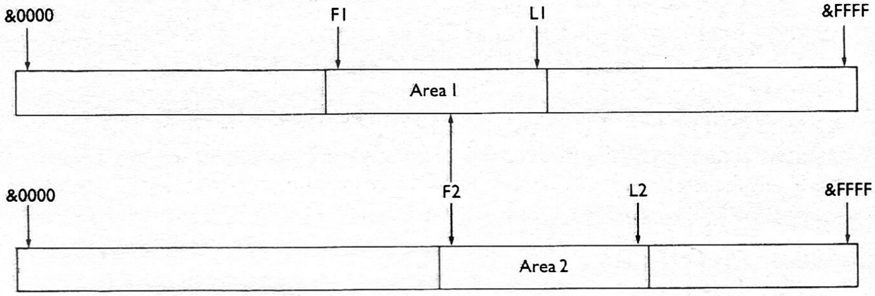
```
   10 REM BLOCK MOVE
   20 PROCaddr("Start byte")
   30 froms%=add%
   40 REPEAT
   50 PROCaddr("Final byte")
   60 fromf%=add%
   70 UNTIL fromf%>froms%
   80 PROCaddr("New  start")
   90 tos%=add%
  100 tof%=tos%+fromf%-froms%
  110 IF froms%<tos% PROCup ELSE PROCdow
n
  120 VDU7
  130 END
  140
  150 DEFPROCaddr(A$)
  160 PRINT'A$;
  170 INPUT "  "X$
  180 add%=EVAL("&"+X$)
  190 ENDPROC
  200
  210 DEFPROCup
  220 REPEAT
  230 ?tof%=?fromf%
  240 tof%=tof%-1:fromf%=fromf%-1
  250 UNTIL tos%>tof%
  260 ENDPROC
  270
  280 DEFPROCdown
  290 REPEAT
  300 ?tos%=?froms%
  310 tos%=tos%+1:froms%=froms%+1
  320 UNTIL tos%>tof%
  330 ENDPROC
```

How it works

The bytes are moved ('copied' is perhaps a more realistic description) one at a time using the '?' (query) operator. It is much faster to use '!'

(pling), which moves four bytes at once, but you would have to know that you were moving a multiple of four bytes, which would not always be the case. Pling could be used to shift the bulk of the bytes and then query could move the last few so that exactly the right number are moved. This is very messy and is hardly worth including for the gain in speed acquired.

If the block is to be moved downwards, the operation takes place from the front and, if upwards, from the back – as in the screen example given earlier. This is absolutely essential so that the block does not get corrupted during the move. If the 'to' and 'from' areas do not overlap, then it is not important how the move is done, but obviously we should cater for all cases. Consider the two memory maps in the diagram:



Here, F1 and L1 are the addresses of the first and last bytes of Area 1 and similarly for F2, L2 and Area 2. The two areas are the same size, namely (L1–F1+1) bytes long.

Suppose that we wish to move Area 1 to Area 2. It should be clear that we cannot start by moving the bytes at F1 to F2 as that would overwrite the byte at F2 within Area 1. The bytes have to be moved in the order:

L1 to L2
L1 – 1 to L2 – 1
L1 – 2 to L2 – 2
. . . . . .
. . . . . .
F1 to F2

Likewise, if Area 2 is to be shifted to overlay Area 1, the first byte to move would be F2 to F1 and the last would be L2 to L1. Before the move starts the routine needs to know which of these two cases it is dealing with so that it can take the appropriate action.

Procedures

PROCaddr prompts with the parameter A$ and accepts an address which it stores in variable ADD%. Two separate routines are required to do the move depending on whether the move is up or down the memory and this is the function of PROCup and PROCdown. Only one of these is used during a RUN of the utility.

Variables

The diagram shows that the important variables in the move routine are the addresses of the start and finish bytes of the area moved from (froms% and fromf%) and those same addresses in the area moved to (tos% and tof%). Actually, one of these is redundant as it can be calculated from the relationship:

fromf%−froms%=tof%−tos%

if we know the other three items – that is why only three addresses are requested by the program.

Extensions

There's not really much to add to this – the most significant change would be to remove some coding if the utility was to be used for a specific task. If you are lucky enough to own discs, then some form of move routine is needed so that programs can be LOADed from disc and then shifted down to &E00, as though they had LOADed from tape. To do this job a very specialised block move is needed in which:

i) The movement is always downwards
ii) tos% probably=&E00
iii) froms% probably=PAGE for your disc system
iii) Pling (!) can be used to speed up the move

The whole routine could then be fitted on a function key, leaving you with something like:

```
*KEY0 I."Last byte",L$:L%=EV.("&"+L$):F.
I%=&E00 TOL% S.4:!I%=I%!&B00:N.¦M
```

Cryptic, but effective.

Utility 6:

String search

Description

This routine locates each occurrence of a string of bytes in memory and prints out its location – i.e. the address of the first byte of the string. Here we are using 'string' in the sense of 'sequence of bytes' rather than the idea of a BASIC string, either A$ or $A. The maximum length of the search string is 32 bytes and it may consist of *any* bytes – not just printable characters.

It can be used with BASIC programs to find all references to a variable or, more usefully, with machine code routines to locate a piece of code. Its use is not limited to programs and it could be used with, say, a text file to locate each occurrence of a particular word.

Two versions are given: the first, in BASIC, should be easy to follow and will enable you to understand the principles involved. The second, in machine-code, is the official version and extremely fast. This routine has been coded to imitate the BASIC program (as far as is practicable) and it is useful to compare the two programs.

Generally speaking, where a machine code routine is needed (either for speed or for compactness) it is good practise to code it into BASIC first to check that the logic of the program is correct. Once the routine has been debugged and tidied-up it can then be re-coded into assembler. Certainly it is not practical to write a machine code routine to mimic the BASIC program line for line but, if you stick to simple BASIC commands, the correspondance between the two will be quite high. With experience, it is possible to develop a BASIC style that produces easily converted code knowing that it is being written for just this purpose. This highlights one disadvantage of the technique: you have to make compromises in the way you initially code the program. Similarly, the machine code that results is not necessarily optimal (in terms of speed) but, if the BASIC has been thought out carefully, it can be close.

Use

Run the program and enter the search string in response to the '?' prompt. If you wish to locate a string of hex bytes, for example every reference to &20EEFF (which is how JSR &FFEE looks in machine code) you should prefix the string with '&' and enter each hex byte as two hex digits – i.e. in the form just quoted.

The routine now prints out all references to that string anywhere in memory (both RAM and ROM). In BASIC, this takes quite a while, so you will have to be patient. To speed things up you could limit the search to a particular area of memory by adjusting the range of loc% in line 80.

If '&' is the first byte of the string you are looking for, then clearly the routine will interpret your string as being hex and to avoid this, you should enter the whole string in hex, with '&' being replaced by &25. For example, to search for the character string '&Y' you should look for the hex string &2659.

When you look for a character string you may find references to it at &3E0–&3FF, which is the keyboard buffer area and in MODE 7, it will appear on the screen, too (between &7C00 and &7FFF). Furthermore, if you run the BASIC program it will appear as a variable at the end of your program. These are unofficial sightings and may be ignored.

When the BASIC version was used to search for the word 'BASIC', the search took 156 seconds, which is reasonably fast for BASIC. However, the machine code routine found all three references in 0.8 seconds (a speed increase of almost 200 times), which says something for the efficiency of machine code.

As written, the assembler version generates the machine code each time it is RUN and this is not necessary as the code is the same each time. If required, you can delete the assembly procedure after the first run, leaving a much shorter, neater procedure.

To summarise, the preferred use of this routine is as follows:

i) LOAD the machine code version at PAGE.
ii) RUN it.
iii) DELETE line 20 and lines 100-460
iv) Set PAGE=PAGE+&100

To use the routine, simply reset PAGE and RUN.

```
   10 REM STRING SEARCH - BASIC
   20 DIM A%(31)
   30 INPUT LINE',A$
   40 L%=LENA$-1
   50 IF LEFT$(A$,1)="&" PROChex ELSE PR
OCchar
   60
   70 match%=0
   80 FOR loc%=0 TO &FFFF
   90 IF ?loc%=A%(match%) PROCfound ELSE
 match%=0
  100 NEXT
  110 END
  120
  130 DEFPROCfound
  140 IF match%=L% PRINT ~(loc%-L%):matc
h%=-1
  150 match%=match%+1
  160 ENDPROC
  170
  180 DEFPROChex
  190 L%=L%/2-1
  200 FOR I%=0 TO L%
  210 A%(I%)=EVAL("&"+MID$(A$,2*I%+2,2))
  220 NEXT
  230 ENDPROC
  240
  250 DEFPROCchar
  260 FOR I%=0 TO L%
  270 A%(I%)=ASCMID$(A$,I%+1,1)
  280 NEXT
  290 ENDPROC
```

```
   10 REM STRING SEARCH (MACHINE CODE)
   20 PROCass
   30 INPUT LINE',A$
   40 L%=LENA$-1
   50 IF LEFT$(A$,1)="&" PROChex ELSE $&
A00=A$
   60 ?&82=L%
   70 CALL &900
   80 PRINT:VDU 7
   90 END
  100
  110 DEFPROCass
  120 addr=&8570 : REM FOR BASIC 2 USE a
ddr=&B545
  130 addl=&80:addh=&81:leng=&82
  140 FOR I%=0 TO 2 STEP 2
  150 P%=&900
  160 [OPTI%:
  170 .L0    LDX #0         ;match=0
  180        STX addl       ;loc=0
  190        STX addh
  200        LDY #0
  210 .L1    LDA (addl),Y ;get a byte
  220        CMP &A00,X   ;found??
  230        BEQ L4       ;yes - 'PROCfou
nd'
  240 .L2    LDX #255     ;no - match=-1
  250 .L3    INX          ;match=match+1
  260        INC addl     ;loc=loc+1
  270        BNE L1
  280        INC addh
  290        BNE L1       ;not finished s
o loop back
  300        RTS          ;to BASIC
  310 .L4    CPX leng     ;whole string m
atched??
  320        BCC L3       ;no - keep chec
king
  330        LDA addl     ;WHOOPEE - it a
ll matches!!!!
  340        SBC leng     ;calculate addr
. of first byte.....
  350        PHA
  360        LDA addh
  370        SBC #0
  380        JSR addr     ;...and print i
t in hex....
  390        PLA
  400        JSR addr     ;....in two ins
talments
  410        LDA #32      ;space
  420        JSR &FFEE
  430        BNE L2       ;unconditional
branch
  440 :]
  450 NEXT
  460 ENDPROC
  470
  480 DEFPROChex
  490 L%=L%/2-1
  500 FOR I%=0 TO L%
  510 I%?&A00=EVAL("&"+MID$(A$,2*I%+2,2)
)
  520 NEXT
  530 ENDPROC
```

How it works

To understand the principles involved, let us consider the operation of the BASIC version of this utility.

Having accepted the search string, A$, array A%(31) is built up from the ASCII codes of the bytes of the string. If A$ is preceded by '&' then the ASCII code is the evaluation of two hex digits in string A$. The size of the array limits the search string to a maximum of 32 bytes.

Memory is scanned one byte at a time, looking for a match with the first byte of the search string. When a match is found an indicator is set to point to the next byte of the string (that is, the next member of array A%(31)) and the search continues. When the indicator shows that all bytes of the string have been matched, the address is printed out. As soon as a comparison fails, the indicator is reset to 0 before the search continues.

This process continues until every byte of memory has been considered as a possible starting point for the string.

Procedures

Each time the indicated byte of the search string matches the current byte of memory, PROCfound is called to check if the whole string has been matched, in which case it prints out the correct address. Otherwise, the indicator is updated to point to the next byte of the string.

When the string is first entered, it has to be analysed and placed into array A%(31) – how this is done depends on whether the string started with a '&' character; if so PROChex will set up the array and if not, PROCchar is called.

Variables

For compatibility between the two types of search string, array A%(31) contains the ASCII codes of the sequence of bytes which is being sought out. Clearly, if the hex option is used, these bytes may well be outside the ASCII range.

loc% is the address of the byte currently under investigation and will assume all values from 0 to &7FFF inclusive.

match% indicates the number of consecutive bytes of the string that have been matched to date and when it reaches the length of the string it indicates that the whole string has matched. It is the pointer into the array A% that was mentioned earlier.

The assembler routine works in almost exactly the same way although the small details are necessarily different. Now the search string is located at &A00 and its length less 1 is stored in location &82, where it can be accessed by the machine code. Variable loc% is stored in two bytes (binary) at &80 and &81, and the variable match% is held in the X-register.

To print out the hex address, the routine makes two calls into the BASIC ROM at &8570 which contains a section of code to print out the contents of the accumulator in hex. We could do this ourselves but, as the code is already there, we may as well use it. If your computer is fitted with BASIC 2, the code will be located at &B545 and you will have to change these calls accordingly. Have a look at this area with your disassembler to find out how it's done.

Extensions

We have already seen how to compress the routine and you are not likely to want to speed it up, so what can we do with it? Some string search routines wait for you to press a key after each occurrence has been identified and before going on to the next one; you can then ESCAPE or carry on with the search. Obviously the idea of this is to enable you to find the significant appearances of the string but a better way is to limit the scope of the search before it starts. You may well feel, as I do, that such an amendment is possible but not really useful as it would probably slow down the search in most circumstances.

One useful, though much more extensive, modification would be to set each appearance in context by printing out the surrounding bytes in both hex and ASCII. At this point we would have included so many extra facilities that the routine may as well be combined with the memory display utility that we have already seen.

Section 2:
BASIC utilities

Each of the utilities in this section modifies or operates upon a BASIC program stored in the computer's memory. To understand how each routine works it is necessary to know how BASIC programs are held in the machine and so we will look at this in some detail.

This is a very valuable exercise and if you have never done so before you should LOAD a program into memory and study it carefully using the memory display utility (in fact you can use the utility to look at itself, if you like). Fig. 1 overleaf shows how the first program of this section looks when LOADed into store at location &1100.

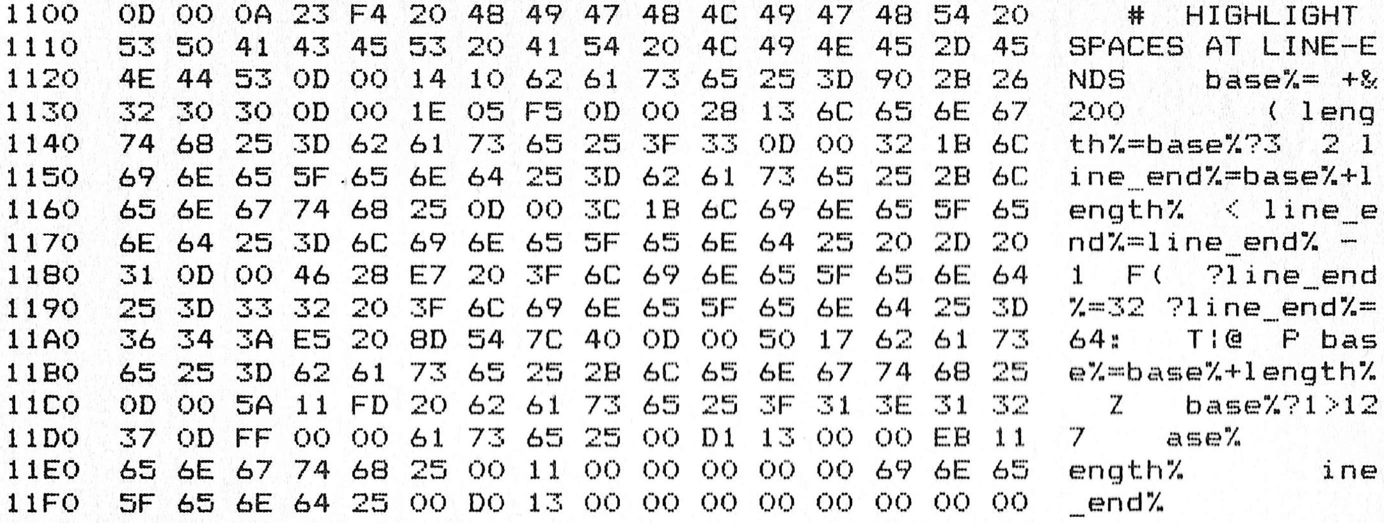
We can now see how the computer sees the program, as opposed to the neat version we see after LIST. Referring to Fig. 1, notice that each line starts with a RETURN character (&0D), and that the following two bytes contain the line number, stored in binary, with the high (*256) byte first. The first &0D character lies at the location PAGE.

The fourth byte of the line is its total length in binary. The length of the line is the displacement from the opening &0D character to the next &0D – that is, it is a pointer to the following line. Thus, to read through the lines – for example, to count them – we only need to start at PAGE and follow the series of links through to the end.

End? Well, the program has to end somewhere and this is indicated by the first byte of the line number being &FF. In Fig. 1 this is located at &11D2. Actually, any byte greater than 127 in this position will serve to terminate the program. The byte following &FF is designated TOP and is usually the start of the variable storage area. TOP is affected by commands such as LIST and END, although it cannot be altered directly. RUNning a program effectively sets LOMEM=TOP so if you want your program to store its variables elsewhere you should reset LOMEM right at the start of your program.

It should now be obvious that the BASIC command NEW merely overwrites PAGE with &0D and PAGE+1 with &FF, whereas OLD resets PAGE+1 back to &00.

From the fifth byte onwards the text is stored in a mixture of 'tokenised' and ASCII formats. When the line is entered at the keyboard it is scanned for BASIC keywords such as REM and PRINT and when these are found, they are replaced by a one byte token. A list of these tokens can be found just before the appendix in the *User Guide*.

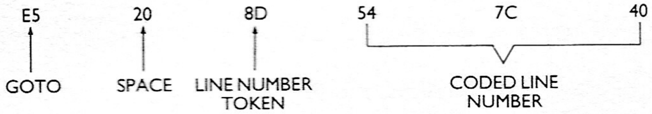


```
1100   0D 00 0A 23 F4 20 48 49 47 48 4C 49 47 48 54 20      #  HIGHLIGHT 
1110   53 50 41 43 45 53 20 41 54 20 4C 49 4E 45 2D 45   SPACES AT LINE-E
1120   4E 44 53 0D 00 14 10 62 61 73 65 25 3D 90 2B 26   NDS    base%= +&
1130   32 30 30 0D 00 1E 05 F5 0D 00 28 13 6C 65 6E 67   200       ( leng
1140   74 68 25 3D 62 61 73 65 25 3F 33 0D 00 32 1B 6C   th%=base%?3  2 l
1150   69 6E 65 5F 65 6E 64 25 3D 62 61 73 65 25 2B 6C   ine_end%=base%+l
1160   65 6E 67 74 68 25 0D 00 3C 1B 6C 69 6E 65 5F 65   ength%  < line_e
1170   6E 64 25 3D 6C 69 6E 65 5F 65 6E 64 25 20 2D 20   nd%=line_end% - 
1180   31 0D 00 46 28 E7 20 3F 6C 69 6E 65 5F 65 6E 64   1  F(  ?line_end
1190   25 3D 33 32 20 3F 6C 69 6E 65 5F 65 6E 64 25 3D   %=32 ?line_end%=
11A0   36 34 3A E5 20 8D 54 7C 40 0D 00 50 17 62 61 73   64:   T¦@  P bas
11B0   65 25 3D 62 61 73 65 25 2B 6C 65 6E 67 74 68 25   e%=base%+length%
11C0   0D 00 5A 11 FD 20 62 61 73 65 25 3F 31 3E 31 32     Z   base%?1>12
11D0   37 0D FF 00 00 61 73 65 25 00 D1 13 00 00 EB 11   7    ase%       
11E0   65 6E 67 74 68 25 00 11 00 00 00 00 00 69 6E 65   ength%       ine
11F0   5F 65 6E 64 25 00 D0 13 00 00 00 00 00 00 00 00   _end%           
```

Fig. 1. How a BASIC program looks to the computer.
This printout was produced by the Memory Dump utility in the previous Section.

Each token is greater than 127 (or in machine language, it is negative!), so that they are not confused with the ASCII characters which make up the rest of the line. Compare the listing of the first utility with Fig.1 to see how most of the text has been tokenised.

One other interesting feature of Fig. 1 is the way in which the instruction: GOTO 60 (in line 70) is stored. It is held at location &11A3 in the form:



We see that the line number is preceded by the token &8D which effectively says 'what follows is an encoded line number' and that the number itself occupies three bytes; this is true whatever the number is. This method of storing line numbers is convenient because:

i) It is easily recognised as being a line number by its token.

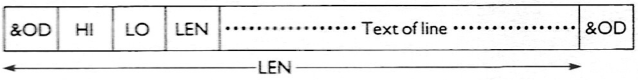
ii) The number always occupies exactly three bytes, however the program is renumbered – replacing GOTO 1 by GOTO 10000 will not extend the program; this is one of the reasons why RENUMBER is so fast.

iii) The number is coded in such a way that the bytes cannot be confused with tokens; in other words, they are all less than 128.

A further point to note from Fig. 1 is that the program has actually RUN and this causes the variables generated by it to be stored at LOMEM, which is usually directly after the program. This causes an overhead when RUNning BASIC programs, which explains why a long program can run out of memory while it is running even though it loads correctly. The last byte assigned to the program in Fig. 1 is at &11F9. We shall look more closely at the method of variable storage later.

The way in which programs are stored is so fundamental to the routines in this chapter that it is worth reviewing the important points here.

i) Starting at PAGE each line is stored in the form:



where the line number is 256*HI+LO and LEN is the number of bytes assigned to the whole line as stored in memory.

ii) The text within the line is held in a mixture of ASCII and (easily recognised) tokens and referenced line numbers.

iii) The last line of the program is stored as:

&OD	&FF

This line is not printed in a LIST – it simply flags the end of the program.

Bearing these points in mind, let us look at a few utilities.

Utility 7:

Highlight end-of-line spaces

Description

This is not a practical utility, but is included here to illustrate some of the observations we have just made.

During program editing using the COPY key it is very easy to copy in extra spaces at the end of a line. These spaces are duly stored along with the rest of the line and use up more memory than is really necessary. Also, it is a common practice (at least in this book!) to include the odd blank line here and there to split the program up into logical sections. It is not obvious just how many spaces have been included in such a line.

This utility spots these extraneous spaces and highlights them by overwriting them with @ characters. Notice that this may render the program unusable by introducing syntax errors – I did warn you it wasn't practical! The next utility is an extension of this one and does actually remove them.

Use

LOAD the utility in the normal way and then set PAGE=PAGE+&200 before LOADing the target program ('victim' might be more appropriate in this case). Now generate some spaces at the ends of a few lines – there may even be some there already. Restore the original value of PAGE and run the highlighter. LISTing the target program (don't forget to change PAGE again!) should reveal that the trailing spaces have been replaced by @'s.

A number of utilities in this section require frequent PAGE changes and it may be worthwhile setting up a couple of function keys to do this for you.

```
   10 REM HIGHLIGHT SPACES AT LINE-ENDS
   20 base%=PAGE+&200
   30 REPEAT
   40 length%=base%?3
   50 line_end%=base%+length%
   60 line_end%=line_end% - 1
   70 IF ?line_end%=32 ?line_end%=64:GOT
O 60
   80 base%=base%+length%
   90 UNTIL base%?1>127
```

How it works

Using the line length as a displacement from the initial &0D we can easily locate the end of the line. Working backwards, we replace any spaces until a non-space character is found. After processing a line, the pointer to the start-of-line is updated and the cycle repeated until the end of the program is reached.

Variables

base% is the address of the &0D character at the start of a line. length% is the length of the line and line_end% is the address of its last byte.

Extensions

An obvious modification to this program involves the development of a routine which actually removes the spaces once they have been identified. This means adjusting the line lengths and shuffling the lines down as spaces are removed. The next utility achieves this.

Utility 8:

Remove end-of-line spaces

Description

This utility removes trailing spaces from lines in the manner just described.

Use

Because the routine is an extension of the previous one, it may be used in exactly the same way.

```
 10 REMOVE SPACES FROM ENDS OF LINES
 20 from%=PAGE+&200:to%=from%
 30 REPEAT
 40 PROCcalc
 50 PROCshuffle
 60 UNTIL from%?1>127
 70 ?to%=13:to%?1=255
 80 END
 90
100 DEFPROCcalc
110 oldlength%=from%?3
120 newlength%=oldlength%
130 REPEAT
140 newlength%=newlength%-1
150 UNTIL from%?newlength%<>32
160 newlength%=newlength%+1
170 ENDPROC
180
190 DEFPROCshuffle
200 FOR I%=0 TO newlength%-1
210 to%?I%=from%?I%
220 NEXT
230 to%?3=newlength%
240 from%=from%+oldlength%
250 to%=to%+newlength%
260 ENDPROC
```

How it works

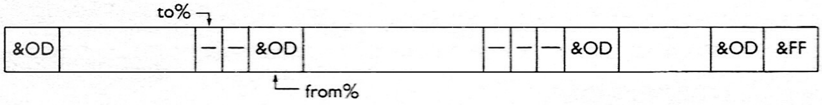
Having found a line with trailing spaces, the line length is recalculated and the rest of the program dropped to overwrite them. Subsequent lines are processed whilst they are shuffled down.

Procedures

PROCcalc reads the length of the line and computes its new length once spaces have been removed. PROCshuffle actually moves the line down into place and stores its new length.

Variables

to% and from% contain the addresses of the beginnings of lines in the 'new' program and the 'old' respectively. Consider the program in the diagram:



After the first line has been processed (notice that it will not get shuffled down as it has nowhere to go), the pointers to% and from% will be as shown. Here, from% points to the actual line while to% points to its intended destination.

When the utility has RUN, from%–to% will be the number of spaces removed from the program.

Extensions

The next routine continues the theme of removing surplus bytes from programs.

Utility 9:

Pack

Description

During the development of long programs it is customary to include a plethora of REMS and spaces so that the code is easy to read. Once the program has been tested, these are no longer necessary; indeed, if memory is short, they can become a liability. Whereas program clarity is a great virtue, it often has to lose out to the more mundane considerations of efficiency and practicality. As a result, the final working version of a program is not always as well documented as one would like.

This utility deletes all unnecessary spaces from programs and removes REMS so as to pack the program into the smallest feasible space. Lines consisting entirely of spaces are reduced to a single space and the text following REMS is removed, although the word REM is retained.

You should only PACK finished versions of programs, as the condensed version can be difficult to edit unless you remember to re-insert significant spaces. For example V=W OR M will be packed to V=WORM, which the interpreter can still recognise as an OR statement since the logical operator is held in tokenised form. However, if you COPY the line you will introduce a new variable called WORM which your coding is unlikely to recognise (unless you happen to be writing a gardening program). This will then cause a 'No such variable' error.

'Pack' can be very effective, especially on large programs. As an example, it took 31 seconds to remove 204 bytes from the earlier disassembler listing.

Use

'Pack' can be used in a similar way to the previous two utilities. It is small enough to fit into Pages 9 and 10, so that LOADing it with PAGE=&900 will leave it waiting in the machine until you need it.

In line 20, the variable page should be set to point to the start of your program before you commit 'Pack' to memory. Once this has been done, PAGE=&900 followed by RUN will pack your program.

```
10 REM PACK
20 page=&3000
30 basef=page:baset=page
40 pf=3:pt=3
```

```
   50 quote=0:rem=0
   60 !baset=!basef
   70 IF basef?1>127 END
   80 pt=pt+1
   90 pf=pf+1
  100 byte=basef?pf
  110 IF byte=&0D baset?3=pt:basef=basef
+pf:baset=baset+pt:GOTO 40
  120 IF rem GOTO 90
  130 IF byte=&22 quote=1-quote:GOTO 180
  140 IF quote GOTO 180
  150 IF byte=&8D baset!pt=basef!pf:pt=p
t+3:pf=pf+3:GOTO 80
  160 IF byte=&F4 rem=1:GOTO 180
  170 IF byte=&20 GOTO 90
  180 baset?pt=byte: GOTO 80
```

How it works

'Pack' is a good example of bad programming – or so the purists would have you believe – because it contains a number of interlocked loops of the form: IF . . . GOTO in lines 110 to 170. However, for such a short program, this type of coding is justified (indeed, it's the reason the program is so short) and accurately reflects the tricky logical decisions that have to be made. If you go to the trouble of flow-charting these few lines you will see precisely how the program works and why the order of the tests is very important.

Each byte of each line is examined for end of line, quote marks, REM, line number token and space. Flags are set as appropriate i.e. whether inside or outside quotes, and the byte may be either written forward to the condensed version or ignored. Thus if both the flag for the current byte being a space and the flag to indicate this is not inside quotes are set then the space will not be carried forward.

Variables

basef and baset are equivalent to from% and to% in the previous program and are used to point to the current line being condensed and its image after the shifting process. quote and rem are flags which are set when those BASIC words are encountered.

Initially (at the start of a line) each is zero and quote is switched between 0 and 1 each time double-quotes are found. This is obviously necessary to avoid packing innocent data which appears in PRINT strings. Once rem is set (to 1) it remains set for the duration of the line; this effectively means that anything following REM is deleted.

byte is the byte of your program currently being evaluated.

Extensions

If required, 'Pack' could be amended to selectively remove items – e.g. REMS but not spaces or versa vice. Another enhancement would be to remove comments from the assembler. This routine would be rather complicated, since the comments would have to be treated as a combination of REM and quote items, since code following assembler comments *is* significant and a simple rem flag is not sufficient. In this way assembler comments are rather different from BASIC REMS.

We leave this as an exercise for the reader!

Utility 10:
Bad

Description

You may occasionally be afflicted by the 'Bad program' error. This is not a derogatory comment by the computer concerning your programming skills, but an admission by the machine that it is unable to interpret the contents of PAGE onwards as being a valid BASIC program in the format previously noted.

When a program is corrupted it can often be restored to almost full health provided the corruption is not too severe. (For example, overwriting one program by another should be regarded as pretty severe!)

Two forms of corruption are common: namely a line-length (byte 4 of a line, remember) is incorrect or control codes (bytes less than 32) appear in the middle of a program. 'Bad Program' essentially means that the line-lengths do not match up all the way through to &0D &FF – the end of the program.

With any program in memory:

```
?(PAGE+3)=?(PAGE+3)+1
```

followed by LIST will cause 'Bad program', and:

```
?(PAGE+3)=&(PAGE+3)-1
```

will restore the situation. This utility searches your bad program and restores the line-length to what it believes to be the correct values. If the end of the program is missing, it will write its own end (&0DFF) when it can't find one within range of the previous line. This may or may not be the true end of your program.

While this process is going on it is a simple matter to check for the presence of any control codes and to overwrite them with @'s.

Actually, this form of 'corruption' is often deliberate as it can render a program unlistable. To see this, add this line to a program:

```
0 REMA
```

now, `?(PAGE+5)=21` will overwrite the A in the REM with a control code 21 – disable VDU drivers. If you attempt to LIST the program, you will see the REM and precious little else as the screen has been disabled. To get out of this, use ESCAPE. Surprisingly the program will still RUN as normal.

Use

When you get a 'Bad program' it is a good idea to *SAVE it before you do anything else just in case your rescue attempt fails.

Enter MODE 7, set PAGE=HIMEM−&200 and load BAD. Edit line 30 so that addr% is the address of your bad program and RUN. Reset PAGE to point to your program, which should now be LISTable. Spurious characters, including a few @'s, may have appeared and the line numbers might be out of step but at least you are now in a position to edit and SAVE the program. In some cases, the program is restored to its original condition because the only fault was an errant line length.

If the line numbers are seriously affected then a quick RENUMBER should sort them out.

'Bad' is also useful for retrieving valuable programs from tape if they refuse to LOAD due to 'Data errors'. If you have such a program, use *OPT 2,0 before *LOADing; this forces the computer to LOAD the program even if it detects errors. An attempt to LIST will almost certainly give 'Bad program' which can then be eliminated with the utility.

If the program is particularly valuable, repeating the above process will generate a number of different versions from which to piece together a complete listing.

```
 10 REM CURE 'BAD PROGRAM'
 20 REM ASSUME LOADED AT.....
 30 addr%=&1900
 40 ?addr%=&0D
 50 REPEAT
 60 d%=3
 70 d%=d%+1
 80 IF d%<255 GOTO 120
 90 addr%?d%=&0D
100 addr%?(d%+1)=&FF
110 GOTO 160
120 byte%=addr%?d%
130 IF byte%=&0D GOTO 160
140 IF byte%<32 addr%?d%=&40
150 GOTO 70
160 addr%?3=d%
170 addr%=addr%+d%
180 UNTIL addr%?1=&FF
```

How it works

Instead of stepping through the program in the (now) familiar way, this utility searches the lines looking for &0D characters (and removing nasty control codes on the way). When &0D is found it is assumed to be the start of the next line and the length of the previous line is adjusted to point to it. The process continues until end of program is found or forced (this occurring when 'Bad' can't find a &0D to chain to).

Variables

addr% is the base of the line being checked and d% is the displacement of the current byte from it into the line.

Modifications

Bad could be condensed to reside at &900 or even to be made available by pressing a function key.

If end of program is forced because, for example, a &0D character is missing, the cured program might be considerably shorter than the original. In this case, the memory display utility can be used to scan the end of the program to see if inserting a dummy &0D somewhere in the last line will solve the problem. Doing this, then re-RUNning 'Bad' will certainly extend the program. A possible extension to 'Bad' would be an automation of this process.

You will realise that retrieving any bad program involves a certain amount of guess-work and trial-and-error. If the nature of the corruption is known then you're in a good position to restore the program. This routine assumes that a particular type of corruption has occurred and does its best to correct it. Because of the way it works, the worst thing that can happen is that the &0D characters go missing. Although it is possible to write 'Bad' routines to deal with this and other disasters, the only really safe way of rescuing your program is manually, using your memory editor. If your program is so 'bad' that even this doesn't work, then I'm afraid you'll have to face up to the fact that your masterpiece is gone for good.

Prevention, as they say, is better than cure.

Utility 11:

No-colon LISTer

Description

This is a short machine-code utility which enables programs to be LISTed in an easily digested form. In this respect, it is an extension of the LISTO utilities already resident in the machine.

The routine produces a LISTO1 listing except that the statements in a multiple statement line are LISTed beneath each other with the colons omitted – this makes the program very easy to read. A listing of 'Pack' in this format appears below.

```
 10 REM PACK
 20 page=&3000
 30 basef=page
    baset=page
 40 pf=3
    pt=3
 50 quote=0
    rem=0
 60 !baset=!basef
 70 IF basef?1>127 END
 80 pt=pt+1
 90 pf=pf+1
100 byte=basef?pf
110 IF byte=&0D baset?3=pt
    basef=basef+pf
    baset=baset+pt
    GOTO 40
120 IF rem GOTO 90
130 IF byte=&22 quote=1-quote
    GOTO 180
140 IF quote GOTO 180
150 IF byte=&8D baset!pt=basef!pf
    pt=pt+3
    pf=pf+3
    GOTO 80
160 IF byte=&F4 rem=1
    GOTO 180
170 IF byte=&20 GOTO 90
180 baset?pt=byte
     GOTO 80
```

Use

Enter the program (without the assembler comments) and RUN it. This generates code stored at &900 which will stay there until you overwrite it with something else and sets up function key 0 to perform the LISTing. Everything, including LIST, will function as normal until you press f0 which will give you a scrolled LISTing in the new format. Notice that you cannot ESCAPE from this LISTing, so that you have to read it to the end, as ESCAPE has been disabled; it is re-enabled when the listing is complete.

```
   10 REM LIST WITHOUT COLONS
   20 FOR I%=0 TO 2 STEP 2
   30 P%=&900
   40 [OPTI%
   50 .L0 CMP #34    ;quotes?
   60      BNE L1    ;no - try something
else
   70      LDA L4+8  ;yes - change quotes
 flag
   80      EOR #255
   90      STA L4+8
  100      LDA #34   ;load quotes again
  110      BNE L3    ;and print them
  120 .L1 CMP #58    ;is it a colon?
  130      BNE L3    ;no - branch to norm
al OSWRCH routine
  140      BIT L4+8  ;are we in quotes, t
hough?
  150      BMI L3    ;yes - so OSWRCH aga
in
  160      STX L4+9  ;OSWRCH should prese
rve XR
  170      LDX #7
  180 .L2 LDA L4,X   ;print <RETURN>, lin
efeed and indent
  190      JSR L3
  200      DEX
  210      BNE L2
  220      LDX L4+9  ;reload XR
  230 .L3 JMP &0000 ;jump to OSWRCH
  240      NOP:NOP
  250 .L4 NOP        ;data goes here:]
  260 NEXT
  270 FOR I%=0 TO 7
  280 READ L4?I%
  290 NEXT
  300 *KEY0 *FX229,1¦M?&937=0:!&92B=!&20
E:?&20F=&09:?&20E=0¦MLISTO1¦MLI.¦M!&20E=
!&92B¦M*FX229,0¦M
  310 DATA 32,32,32,32,32,32,10,13
```

How it works

Pressing f0 re-directs the OSWRCH vector (OSWRCH is the routine responsible for writing all characters to the screen, and most other places) to point to the code located at &900.

If a ':' is detected, it is suppressed and replaced by a line feed and six spaces. Of course, if the ':' is not a statement separator (in other words it is in quotes) it should be PRINTed in the normal way. Location &937 is used as a quotes flag *à la* 'Pack' (0 means out, 255 means in) and is initialised to 0 on pressing f0. If your program contains an error such that quote marks do not occur in pairs, the routine will go out of synch and produce some strange (but harmless) effects.

Utility 12:

Find

Description

When debugging or amending long programs it is often necessary to know the location of certain variables or constants. This utility PRINTS out the line numbers at which a selected item occurs, together with the total number of occurrences. In fact the 'variable' could equally well be a constant since, as we have seen, they are stored in the same ASCII format. For completeness, the routine can also find each appearance of a BASIC keyword.

Use

The program is easy to use and is best left dormant in the computer whilst debugging your program. To do this, LOAD it in at PAGE and edit line 50 to point at, say, PAGE+&600. This is where the routine expects to find your code, so set PAGE=PAGE+&600.

Programs may now be LOADed and RUN in the usual way. To use the utility reset PAGE and RUN. Input the item you wish to find – either a variable (e.g. BASE%), constant (e.g. 33005), or BASIC keyword (e.g. CALL). The resulting line numbers show where it can be found.

One minor limitation of the routine is that you cannot use it to search for line numbers because, as we know, they are held in encoded form.

```
   10 REM BASIC 'FIND'
   20 REM < 6 PAGES
   30 occ=0:lines=0:token=0
   40 REM **** SET loc TO BASE OF PROG :
 eg...
   50 loc=&1900
   60 INPUT'"Search for "A$
   70 la=LENA$
   80 PROCtesttoken
   90 REPEAT
  100 found=0
  110 IF token PROCkeyword ELSE PROCfind
  120 IF found PROCbaseten
  130 occ=occ+found
  140 loc=loc+loc?3
  150 UNTIL loc?1>127
  160 PRINT 'occ" Occurrences in ";lines
" lines"
  170 END
  180
  190 DEFPROCfind
  200 IF la>loc?3-5 GOTO 360
  210 IF INSTR($(loc+4),A$)=0 GOTO 360
  220 quote=0:rem=0:skip=0:n=1:ptr=4
  230 REPEAT
  240 byte%=loc?ptr
  250 IF byte%=&F4 rem=1
  260 IF rem GOTO 340
  270 IF byte%=&22 quote=1-quote:GOTO 34
0
  280 IF FNtest(byte%)=0 GOTO 330
  290 IF skip OR quote GOTO 340
  300 IF byte%<>ASC(MID$(A$,n,1)) skip=1
:GOTO 330
  310 IF n<la n=n+1:GOTO 340
  320 IF FNtest(loc?(ptr+1))=0 found=fou
nd+1
  330 n=1
  340 ptr=ptr+1
  350 UNTIL ptr>loc?3
  360 ENDPROC
  370
  380 DEFFNtest(B%)
  390 C%=1
  400 IF B%<36 OR B%>122 C%=0
  410 IF B%>38 AND B%<48 C%=0
  420 IF B%>57 AND B%<64 C%=0
  430 IF C%=0 skip=0
  440 =C%
  450
  460 DEFPROCbaseten
  470 PRINT 256*(loc?1)+loc?2
  480 lines=lines+1
  490 ENDPROC
  500
  510 DEFPROCtesttoken
  520 keys=&806B : REM FOR BASIC 2 USE k
eys=&806F
  530 REPEAT
```

```
540 word$="":keys=keys+2
550 REPEAT
560 word$=word$+CHR$(?keys)
570 keys=keys+1
580 UNTIL ?keys>127
590 UNTIL A$=word$ OR word$="WIDTH"
600 IF A$=word$ token=?keys
610 ENDPROC
620
630 DEFPROCkeyword
640 FOR ptr=4 TO loc?3
650 IF loc?ptr=token found=found+1
660 NEXT
670 ENDPROC
```

How it works

Having accepted the item, it is first tested to see whether it is a BASIC keyword and, if so, its token is searched for instead. The lines are studied one at a time by means of the familiar base and displacement technique and if the string appears in a line it is checked to be a valid appearance; this is the most difficult part of the program. Unfortunately we cannot just use the powerful INSTR command as the following example shows.

Consider the line:

INPUT "TYPE A NUMBER" A$: A%=A%+1

A superficial test would suggest that the variable A occurs four times in this line, whereas in fact it is not there at all. The reasons for this should be clear and give rise to much of the coding in PROCfind. A more subtle example is provided by:

FOUR=TWO+THREE (perfectly good BASIC!)

ask yourself why you should *not* find variables such as T and OUR in this line. You should now understand the rest of PROCfind.

Procedures

PROCfind scans the line for the hunted item and may call function TEST to resolve tricky cases like the last example. An initial inspection is made with INSTR and, if fruitful, the PROCedure takes a much closer look at the line. PROCbaseten prints out the line number and keeps a count of the number of line numbers output.

PROCtesttoken is used to check whether or not the item you are seeking is a reserved word. This is possible as BASIC keeps a list of these starting at &806B together with their tokens.* It is well worth having a look at this area of ROM with your memory display utility.

If you *are* looking for a keyword, the coding is considerably simplified, and PROCkeyword will do the job for you.

Variables

token is set to the relevant BASIC token if searching for a keyword, otherwise it is set to 0. loc is the address of the current line in RAM and found is the number of times the item is found on that line.

ptr is the displacement from loc of the current byte being inspected – not surprisingly called byte. quote and rem flags are the same as we used in 'Pack' and skip is a flag set to say 'even if you do find the string, ignore it'. The purpose of this flag is to avoid variables that appear at the end of strings, such as the OUR in the last example quoted. skip is reset each time a byte that cannot be part of a string is detected.

Extensions

'Find' could be combined with an editor which actually listed the entire line rather than just its number. This would set the item in context and make it easy to skip through the occurrences when searching for the ones required.

One very useful addition would allow the variable to be replaced by another – this is a great boon in development work and is the subject of the next utility.

*Note: If your machine has BASIC 2, the keywords start at &806F and PROCkeys should be set accordingly.

Utility 13:

Replace

Description

One of the admirable features of the BBC Micro is its ability to handle variable names of any length. This means that programs may feature meaningful labels such as loop-count or answer$ instead of the traditional and rather laconic I% or A$. However, should a program get too long (yes, that ever present problem of limited memory again) it is sometimes prudent to trim some of your more verbose labels down by a few bytes. Alternatively, thrifty programmers used to working with very short names might want to expand them to make the program more readable.

This utility is a cannibalised version of the previous one and enables items, once located, to be replaced by a different name (or constant, if you wish).

Use

The routine is used in the same way as 'Find' except that two names are entered: the item to be replaced, and its substitute. Each time a replacement occurs the machine 'blips' just to let you know that it really is doing something since no results are printed out at the end.

Use of this routine may cause your program to expand and, like the utility that follows, it has to be very careful where it stores BASIC variables so that they are not overwritten. In line 30, LOMEM is set to PAGE+&490 which is just past the end of the utility, and your program should start no earlier than PAGE+&600 (as with 'Find').

When you have finished keying in the routine, check the value of ~(TOP-PAGE). If it is greater than &490, then you have copied in more spaces than you need. Either remove them, or adjust line 30 accordingly.

```
   10 REM BASIC 'REPLACE'
   20 REM < 6 PAGES
   30 LOMEM=PAGE+&490
   40 REM ***** SET loc TO BASE OF PROG
: eg...
   50 loc=&3000:top=loc
   60
   70 REPEAT
   80 top=top+top?3
   90 UNTIL top?1>127
  100 top=top+2
  110
  120 INPUT'"Search for "A$
  130 INPUT'"Replace by "B$
  140 la=LENA$:lb=LENB$:diff=lb-la
  150
  160 REPEAT
  170 PROCfind
  180 loc=loc+loc?3
  190 UNTIL loc?1>127
  200 END
  210
  220 DEFPROCfind
  230 IF la>loc?3-5 GOTO 390
  240 IF INSTR($(loc+4),A$)=0 GOTO 390
  250 quote=0:rem=0:skip=0:n=1:ptr=4
  260 REPEAT
  270 byte%=loc?ptr
  280 IF byte%=&F4 rem=1
  290 IF rem GOTO 370
  300 IF byte%=&22 quote=1-quote:GOTO 37
0
  310 IF FNtest(byte%)=0 GOTO 360
  320 IF skip OR quote GOTO 370
  330 IF byte%<>ASC(MID$(A$,n,1)) skip=1
:GOTO 360
  340 IF n<la n=n+1:GOTO 370
  350 IF FNtest(loc?(ptr+1))=0 PROCrepla
ce
  360 n=1
  370 ptr=ptr+1
  380 UNTIL ptr>loc?3
  390 ENDPROC
  400
  410 DEFFNtest(B%)
  420 C%=1
  430 IF B%<36 OR B%>122 C%=0
  440 IF B%>38 AND B%<48 C%=0
  450 IF B%>57 AND B%<64 C%=0
  460 IF C%=0 skip=0
  470 =C%
  480
  490 DEFPROCreplace
```

```
500 start=ptr-la+1
510 IF diff<0 PROCdown
520 IF diff>0 PROCup
530 FOR I%=1 TO lb
540 loc?(start+I%-1)=ASC(MID$(B$,I%,1)
)
550 NEXT
560 top=top+diff:ptr=ptr+diff
570 loc?3=loc?3+diff
580 SOUND 1,-8,200,1
590 ENDPROC
600
610 DEFPROCdown
620 B%=loc+start+la
630 REPEAT
640 B%?diff=?B%:B%=B%+1
650 UNTIL B%>top
660 ENDPROC
670
680 DEFPROCup
690 B%=top
700 REPEAT
710 B%?diff=?B%:B%=B%-1
720 UNTIL B%<loc+start
730 ENDPROC
```

How it works

Because this routine has to shift your program around in memory, it must first calculate the position of the TOP of your program before it can enter its main loop.

Using PROCfind as in the previous utility, the program scans the lines looking for the item to be replaced. When found, it opens or closes a gap in the program, depending on the relative sizes of the two names, and writes in the new name. Although this process is easy to understand, it involves a great deal of memory manipulation. Despite this, the routine is still quite fast.

Procedures

PROCfind and PROCtest have been borrowed from 'Find' and have been coded to perform the same function.

PROCreplace is called whenver the sought item is located. If the replacing string is shorter than the replaced string then PROCdown is called to shuffle the top part of your program down the memory before the string is written into place. If longer, PROCup is called to shift your program up in memory so as to open a gap to insert the new name. Obviously neither needs to be called if the two strings are the same length. In this case, the replacement is very rapid.

Variables

top always points to the TOP of your program and keeps track of it while your program is being shoved around. 1a and 1b are the lengths of the two strings and diff is the difference between them. start points to the first byte of the string being replaced.

Extensions

As this routine has so much in common with 'Find', they could be incorporated into one long utility. Even if you don't do this it is worth typing them in together and COPYing common chunks of code.

Stingy programmers sometimes use the same variable for more than one purpose in different parts of a program – see how B% is used in this very routine! You may wish to extend 'Replace' so that only selective replacement takes place whereby you are offered each occurrence and may choose whether or not to replace it.

Finally, the routine will fail if a line grows to more than 255 characters. This occasionally happens if you replace a short variable name in a long line with something-of-this-sort-of-size. It is possible to detect this and to prevent it happening – another exercise for the reader. Personally, I think you deserve all that you get if you enter lines of this sort of length!

Utility 14:

MODE 7 PROC writer

Description

We now look at a very interesting utility which is both useful in itself and also representative of a common and rather important type of utility.

This is a program which actually writes BASIC! What it produces is one (or several) PROCedures tagged on to the end of your own program to format MODE 7 screens.

MODE 7 is commonly used to display rules, menus (not necessarily gastronomic), instructions, lists and the like. The coding associated with such activities is invariably rather dull, even – dare I say it – boring. Now one of the uses of a computer is to take over those human tasks which we find tedious and screen formatting procedures definitely fall into this category.

With 'PROC Writer' you design the screens and the computer writes the code for you (and quite a neat little coder it is, too). If the idea of computers writing their own programs gives you a Big-Brother complex, let us assure you that we did actually write this routine by hand. Not many of the utilities in this book were written by computers!

Using the cursor keys for movement, data is entered on to a blank MODE 7 screen. When ESCAPE is pressed, the screen is read and a PROCedure is generated to produce that screen. This PROCedure will appear at the end of your own program in store. You then have the option of formatting further screens (up to 99) or quitting. PROCedures so generated are separated from your program and each other by a blank line and are called 'screen1', 'screen2' and so on. The line numbers assigned to the PROCedures follow on from your program in steps of 10.

We are already familiar with the format of BASIC lines and this in itself is not too difficult to mimic. The complication lies in producing valid, syntactically correct BASIC and this could take us into all sorts of areas including logic and semantics. You should find it a very rewarding exercise to extend this utility and to produce others in the same vein. The idea is to select a 'vocabulary' and devise rules of 'grammar' which effectively determine how the elements of the vocabulary should interact.

For the purposes of this utility the vocabulary is made up of the following items:

DEF PROC CLS PRINT screen
TAB(ENDPROC 0–9) ' " ; ,

plus the ASCII data you enter on to the screen.

We now have to sort this lot into a meaningful order and the rules for this are fairly straightforward. For example: CLS occupies a line on its own, PROC follows DEF, ' can only follow PRINT or itself and so on.

The resulting PROCedure can be called by your program and is written in reasonably efficient code.

Use

LOAD the utility, set PAGE=PAGE+&A00 and then LOAD in the receiving program. Type END <RETURN> and then T%=TOP <RETURN>. Return (that word again!) to the utility (remembering to reset PAGE), and RUN it.

Enter whatever data you like on the screen using the cursor keys for rapid movement and SHIFT-function key combinations to produce colour control codes. If you wish to access other codes, use *FX226 or *FX227 to reset the 'base' for SHIFT and CTRL function keys respectively. See the *User Guide* for more details. DELETE works in the usual way or, alternatively, incorrect items may be overwritten with new data.

When you are happy with the screen press ESCAPE. The utility will read the screen and generate the PROCedure to reproduce it. When the computer bleeps, press Y to create another PROCedure or N to end.

When you have finished, restore PAGE and study your original program. It will include one or more new PROCedures which you can call from within your program with PROCscreen1, PROCscreen2 etc. Any colour or other control codes are copied straight from the screen into the PRINT string rather than having them produced with PRINT CHR$ instructions; this makes the coding easier, and the finished PROcedure somewhat shorter.

```
 10 REM MODE 7 PROC WRITER
 20 MODE 7
 30 *FX 4,1
 40 *FX12,5
 50 LOMEM=PAGE+&840
 60 T%=T%-2
 70 screen%=0:base%=T%
 80 REPEAT:T%=T%-1
 90 UNTIL ?T%=&0D AND T%+T%?3=base%
100 lineno%=T%?2+256*(T%?1)
110 ON ERROR GOTO 210
120 REPEAT
```

```
130 F%=INKEY(0)
140 IF F%=136 PROCmove(-1,0)
150 IF F%=137 PROCmove(+1,0)
160 IF F%=138 PROCmove(0,+1)
170 IF F%=139 PROCmove(0,-1)
180 IF F%>31 VDU F%
190 UNTIL FALSE
200
210 ON ERROR GOTO 570
220 lastline%=-1
230 screen%=screen%+1
240
250 PROCstartline
260 PROCstore(&20)
270 PROCendline
280
290 PROCstartline
300 RESTORE
310 FOR I%=1 TO 8
320 READ W%:PROCstore(W%)
330 NEXT
340 PROCsetup(screen%)
350 PROCendline
360
370 PROCstartline
380 PROCstore(&DB)
390 PROCendline
400
410 FOR Y%=0 TO 24
420 PROCscrline
430 NEXT
440
450 PROCstartline
460 PROCstore(&E1)
470 PROCendline
480
490 ?base%=&0D:base%?1=&FF
500
510 *FX 15,1
520 REPEAT
530 VDU 7
540 W%=INKEY(300)
550 UNTIL W%=89 OR W%=78
560 IF W%=89 CLS:GOTO 110
```

```
  570 VDU 30:REPORT:PRINT
  580 *FX 4,0
  590 *FX12,0
  600 END
  610
  620 DEFPROCmove(dx%,dy%)
  630 VDU 31,POS+dx%,VPOS+dy%
  640 F%=0
  650 ENDPROC
  660
  670 DEFPROCscrline
  680 X%=39
  690 REPEAT
  700 PROCreadscrbyte
  710 X%=X%-1
  720 UNTIL X%<0 OR byte%>32
  730 IF X%<0 ENDPROC
  740
  750 last%=X%+1
  760 PROCstartline
  770 PROCstore(&F1)
  780 tabswitch=1
  790 IF Y%<lastline%+5 PROCdownquotes
  800
  810 X%=-1
  820 REPEAT
  830 X%=X%+1
  840 PROCreadscrbyte
  850 UNTIL byte%>32
  860 first%=X%
  870 IF tabswitch PROCxytab:GOTO 890
  880 IF first%>5 PROCxtab ELSE first%=0
  890 PROCstore(&22)
  900 X%=first%
  910 REPEAT
  920 PROCreadscrbyte
  930 PROCstore(byte%)
  940 X%=X%+1
  950 UNTIL X%>last%
  960
  970 PROCstore(&22)
  980 IF (last%=39 OR Y%=24) PROCstore(&
3B)
  990 PROCendline
```

```
1000 lastline%=Y%
1010 ENDPROC
1020
1030 DEFPROCdownquotes
1040 tabswitch=0
1050 REPEAT
1060 PROCstore(&27)
1070 lastline%=lastline%+1
1080 UNTIL Y%=lastline%
1090 ptr%=ptr%-1
1100 ENDPROC
1110
1120 DEFPROCxytab
1130 PROCstore(&8A)
1140 PROCsetup(first%)
1150 PROCstore(&2C)
1160 PROCsetup(Y%)
1170 PROCstore(&29)
1180 ENDPROC
1190
1200 DEFPROCxtab
1210 PROCstore(&8A)
1220 PROCsetup(first%)
1230 PROCstore(&29)
1240 ENDPROC
1250
1260 DEFPROCstartline
1270 lineno%=lineno%+10
1280 ?base%=&0D
1290 base%?1=lineno% DIV 256
1300 base%?2=lineno% MOD 256
1310 ptr%=4
1320 ENDPROC
1330
1340 DEFPROCendline
1350 base%?3=ptr%:base%=base%+ptr%
1360 ENDPROC
1370
1380 DEFPROCsetup(V%)
1390 IF V%>9 PROCstore((V%DIV10)+48)
1400 PROCstore((V%MOD10)+48)
1410 ENDPROC
1420
1430 DEFPROCstore(U%)
```

```
 1440 base%?ptr%=U%:ptr%=ptr%+1
 1450 ENDPROC
 1460
 1470 DEFPROCreadscrbyte
 1480 VDU 31,X%,Y%
 1490 A%=&87
 1500 W%=USR(&FFF4) AND &FFFF
 1510 byte%=W% DIV &100
 1520 ENDPROC
 1530
 1540 DATA &DD,&F2,&73,&63,&72,&65,&65,&
6E
```

How it works

Using the value of T%, the routine first locates the last line number of the target program and then enters a loop (lines 120 – 190) to enable you to set up the screen. By this time the cursor keys have been deprived of their usual function and merely generate ASCII codes. All other keys write data to the screen.

On ESCAPEing from this loop, three lines are added to your program:

i)	A blank line	(Lines 250 – 270)
ii)	DEFPROCscreen1	(Lines 290 – 350)
iii)	CLS	(Lines 370 – 390)

Further lines to produce the required screen are now added – one for each line of the screen on which data occurs, and the process ends with an ENDPROC line, followed by &0D &FF to form a new TOP for your program. Finally, you are offered the option of creating a new PROCedure or completing the run.

This gives the general thread of the program. Like a few other routines in this book, the really clever bits are within the PROCedure following the main body of the program.

Procedures

PROCmove is a simple routine to move the cursor around under the control of the arrowed keys.

Most of the work is done by PROCscrline which is responsible for reading a line of the screen and, if data is found there, writing the relevant coding into your program. The algorithm for this is determined, to some extent, by the 'rules of grammar' mentioned earlier and is depicted in the flow-chart of Fig. 2.

PROCdownquotes stores sufficient ' characters to reach this screen

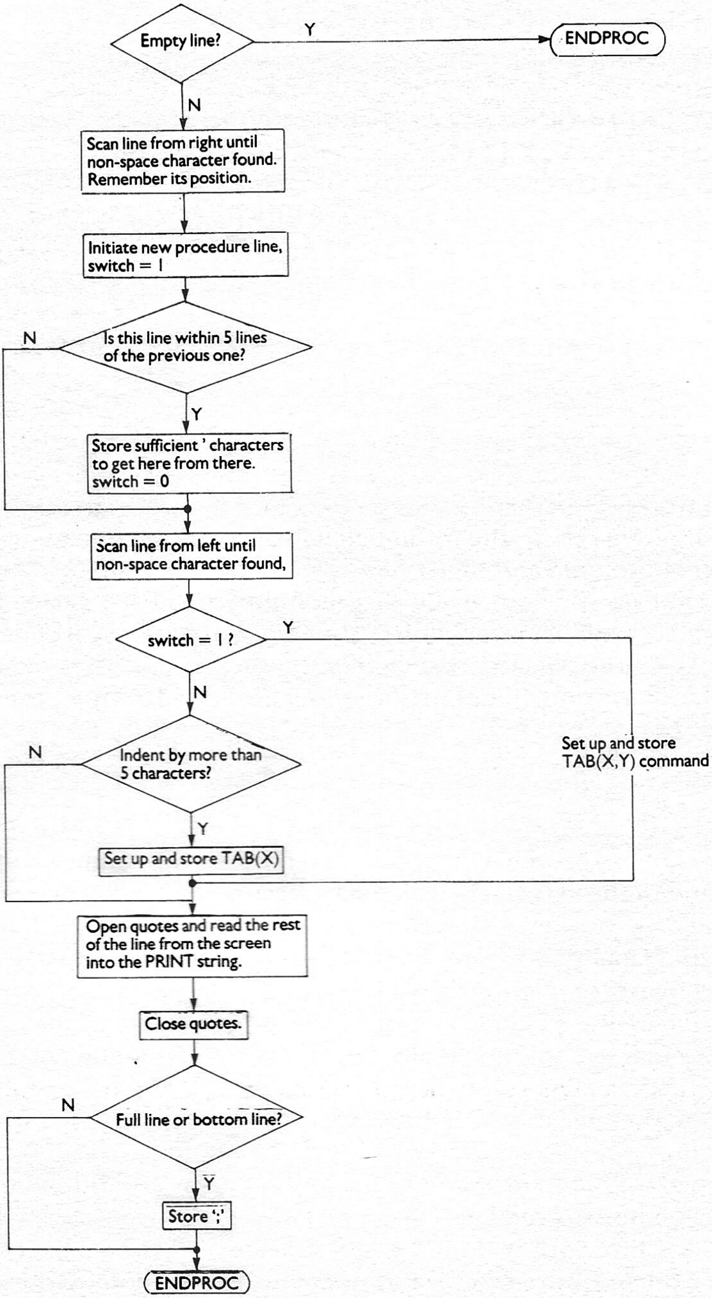

Empty line?
Y
ENDPROC
N
Scan line from right until non-space character found. Remember its position.
Initiate new procedure line, switch = 1
N
Is this line within 5 lines of the previous one?
Y
Store sufficient ' characters to get here from there. switch = 0
Scan line from left until non-space character found,
switch = 1 ?
Y
N
Set up and store TAB(X,Y) command
N
Indent by more than 5 characters?
Y
Set up and store TAB(X)
Open quotes and read the rest of the line from the screen into the PRINT string.
Close quotes.
N
Full line or bottom line?
Y
Store ';'
ENDPROC

Fig 2. Flow diagram of PROCscrline

line from the previous one (on which data occurs). Both PROCxytab and PROCxtab set up TAB(ready to accept either one (X) or two (X,Y) parameters. If more than five spaces are needed to prefix a line, then a TAB(command is used to start the line off. Have a look at the list of BASIC tokens given in the *User Guide* if you want to know how these PROCedures actually get the data into your program.

PROCstartline generates &0D and a line number ready to start a new line in the target program; the line is completed by calling PROCendline which writes in its length and then updates the pointer PROCbase% to point to the next line.

To enable the utility to store numbers (in ASCII format, remember) in your program, the procedure PROCsetup converts the parameter V% into its ASCII equivalent. PROCsetup can handle numbers up to two digits long – i.e. up to 99 – which explains the limit on the number of PROCedures possible. In practise, this should be quite sufficient!

PROCstore is a very important one-liner as it inserts all the data into the lines of your program. The items are passed singly to PROCstore as a parameter U%. Finally, PROCreadscrbyte reads the contents of the screen byte at X%,Y% into the variable PROCbyte% – this is done by using an OSBYTE call with A=135 (see *User Guide)*.

Variables

base% is the address of the &0D (start) of the current line which is being built up in your program. ptr% is the displacement from it of the byte currently being – or about to be – stored.

lastline% is the screen line number (0–24) of the most recent line of the screen on which data is stored, excluding the current one. It is used to decide whether to reach the current line by inserting ' characters or by using a TAB(X,Y) command.

first% and last% are the positions of the first and last non-space bytes on the line being studied, and lie in the range 0–39.

Extensions

There is plenty of scope for experimenting with this utility by extending the facilities available or even by adapting it to write PROCedures (and programs?) of a totally different nature. A reasonably straightforward modification would be to generate PROCedures to format screens in all MODEs, not just MODE 7. To do so you would have to extend the vocabulary to include words like COLOUR and to devise new rules of syntax for those words.

If you do attempt this – and it's a very rewarding exercise – you may begin to appreciate the problems encountered by the people who devise computer languages. Each word that is added to the vocabulary brings with it a new set of rules and, unfortunately, may disturb existing rules which worked fine until the new item was introduced. Before you attempt anything like this let's consider a less demanding extension.

One minor inefficiency of this utility is that each screen line generates a single PRINT string, whatever the format of the data. A line such as:

HELLO <25 SPACES> SAILOR

would generate a 36 byte PRINT string although it could be stored more efficiently as:

PRINT"HELLO"SPC(25)"SAILOR"

In other words, the vocabulary of the routine could be extended to include items such as SPC (CHR$ and ":" are other suitable candidates), according to the requirements of the user.

As a programmer, I have found this routine to be extremely useful and it is a true utility in the sense that we have defined it: a time and labour-saving aid to program development. What more could you want?

Really must get to work on that ultimate utility – the one that writes utilities!

Utility 15:
New TRACE/BASIC single step

Description

The TRACE facility is a useful aid to debugging but the results are often difficult to interpret because the line numbers it generates are interspersed with program output and user input. This tends to defeat the object of program tracing which, after all, is to help you find errors in your program. Most of the people I know who own a BBC Micro avoid this command simply because of the unfriendly nature of the output. This is a pity because TRACE, if properly used, can be a very powerful tool when trouble-shooting on your micro.

'New TRACE' modifies the inbuilt TRACE routine to make it much more usable, printing the generated line numbers at the top left-hand corner of the screen (allright – the current text window) by intercepting the line numbers before they are printed.

While we are at it (intercepting line numbers, that is), having trapped the line number, it is useful to hold the machine up until a key is pressed. In this way, we introduce an exceptionally useful single-step into BASIC which simplifies debugging to a remarkable degree. Despite the power of this facility, the routine that accomplishes it is only just over 100 bytes of machine code!

Use

Enter the routine (omit the assembler comments if you like) and SAVE it. RUNning the program will store the new coding in Page 9, where it should remain until you overwrite it, and function keys f0 and f1 will be set up to switch the new routine in and out. Be careful not to re-program these particular keys during any subsequent playing with the machine. The BASIC coding is now redundant and can be removed with a DELETE or a NEW.

To use with your BASIC program, press f0 and then <RETURN> to activate the new TRACE routine. Now when the program is run, the traced line numbers will appear in the top left-hand corner of the screen and, after each is printed, the computer will wait for you to press the space-bar before proceeding to the next line. Holding down the space-bar will cause your program to run in 'slow-motion', making it very easy to track the lines.

When the run is complete, or after you have ESCAPEd, press f1 to cancel TRACE, and also to return the machine to normal output. Use f0,

<RETURN> any time you wish to TRACE a program and always cancel it with f1 after use.

```
   10 REM NEW TRACE/BASIC SINGLE-STEP
   20 jump=&80:wrch=&81:flag=&83
   30 ptab=&84:xtab=&85:ytab=&86
   40 osbyte=&FFF4:osrdch=&FFE0
   50 FOR I%=0 TO 2 STEP 2
   60 P%=&900
   70 [OPTI%:
   80      BIT &26A      ;vdu queue still
active?
   90      BMI L2        ;yes, so come str
aight out
  100      BIT flag      ;was last byte a
close bracket?
  110      BPL L1        ;no - carry on wi
th test
  120      PHA           ;yes, so skip (th
is) space
  130      LDA #0
  140      STA flag      ;reset flag...
  150      PLA
  160      RTS           ;...and exit
  170
  180 .L1 CMP #91        ;is it a left bra
cket?
  190      BEQ L3        ;yes - branch
  200      CMP #93       ;maybe a close br
acket??
  210      BEQ L4        ;yes - branch
  220 .L2 JMP (wrch)     ;neither so do no
rmal OSWRCH
  230
  240 .L3 PHA            ;left bracket rec
eived
  250      TXA           ;    - OSWRCH shou
ld preserve registers
  260      PHA
  270      TYA
  280      PHA
  290      LDA #134      ;read cursor posi
tion
  300      JSR osbyte
  310      STX xtab        ;and remember it
  320      STY ytab
  330      LDX #4          ;pointer to  i) h
ome cursor ii) print
  340      BNE L7          ;     5 spaces iii
) home cursor again
  350
  360 .L4 PHA              ;right bracket re
ceived
  370      TXA             ;    - preserve al
l registers
  380      PHA
  390      TYA
  400      PHA
  410      LDA #255        ;set right bracke
t received flag
  420      STA flag
  430      LDA #15         ;clear keyboard b
uffer
  440      LDX #1
  450      JSR osbyte
  460 .L5 JSR osrdch       ;get a key
  470      CMP #27         ;ESCAPE?
  480      BNE L6          ;no - branch
  490      LDA #&7E        ;acknowledge ESCA
PE
  500      JSR osbyte
  510      LDA #125        ;and implement it
  520      JSR osbyte
  530      CLV
  540      BVC L8
  550 .L6 CMP #32          ;space bar?
  560      BNE L5          ;no, so get anoth
er key
  570
  580      LDX #0          ;point to PRINT T
AB(..
  590
  600 .L7 LDA ptab,X       ;send bytes to us
ual OSWRCH
  610      BMI L8          ;until negative
  620      JSR (jump)
  630      INX
  640      BNE L7
```

```
 2698
  660 .L8 PLA              ;all done - resto
re registers..
  670     TAY
  680     PLA
  690     TAX
  700     PLA
  710     RTS              ;and return
  720
  730 :]
  740 NEXT
  750 ?&80=&4C:?&81=?&20E:?&82=?&20F
  760 *KEY0 ?&20E=&00:?&20F=&09:TRACE ON
|M
  770 *KEY1 ?&20E=?&81:?&20F=?&82:TRACE
OFF|M
  780 FOR I%=&83 TO &8F
  790 READ ?I% : NEXT
  800 DATA 255,31,0,0,255,30,32,32,32,32
,32,30,255
```

How it works

The routine exploits the (fortuitous – or was it planned?) fact that the traced line numbers are surrounded by [ ] characters (in MODE 7, these appear as left and right arrows, respectively). OSWRCH is the routine responsible for writing characters to the screen and so we intercept OSWRCH and look for these characters (for non-machine coders: this is rather like steaming open someone's mail, only a bit quicker) – their ASCII codes are 91 and 93.

However, it may be that the code is being sent *not* as a result of say:

PRINT "["

but perhaps as part of a string of control codes as in:

VDU 23,224,91,93,0,8,0,8,6,6

In this case we should let the code through unchallenged. This is done by inspecting the VDU queue to see if a command such as the one above is being processed, if so, we ignore the character. The VDU queue tells the machine how many bytes are still expected by OSWRCH before it can go about its normal business of writing characters to the screen, and is located at &26A. Officially, it is considered bad form to read this location directly and there is an OSBYTE call (&DA) to do the job for us. However, as with so many things in life, the temptation is often too great to resist – in our case for the simple reason that the coding is quicker and easier.

When '[' is received, we remember the cursor position (OSBYTE call with A=134) and home the cursor ready to print out the line number. Having done so, a ']' will duly arrive – this is suppressed and a loop entered at L5–L6 waiting for the space-bar or ESCAPE. If the space-bar is pressed the cursor is relocated to its stored position and the program continues.

Simple, isn't it?

Variables

flag is set when a ']' character has just been detected. This is because the normal traced output separates its line numbers with a space, and this space can play havoc with the cursor position which is used by this routine. flag is a marker used to suppress this space.

xtab and ytab are used as a temporary store for the 'real' cursor position.

Extensions

This useful little routine does suffer from a number of drawbacks, none of which are serious, although it is as well to be aware of them.

Any '[' character attempting to reach the screen is assumed to precede a line number and ']' characters are similarly misunderstood. Your program should avoid printing these characters (why can't you use nice round brackets like everyone else?). Also, indiscreet colour changes and 'join cursor' commands can wreck the line numbers output by the routine. It is hardly worth extending the utility to anticipate these problems – it's easier to temporarily modify your program to avoid them.

One other minor point is that, since the line numbers are printed at the top left of the screen, it follows that nothing you print there is going to survive for long, so don't panic if that part of your display disappears.

If you program in BASIC, you will find this utility to be an invaluable debugging aid.

Finally, here is a puzzle for you to see if you have understood this section. LOAD or type in the routine and RUN it. Press f0 and then <RETURN>. So far so good, but why won't the routine list properly?

Think about it.

Utility 16:

Symbol table

Description

Another useful debugging aid is a 'symbol table' – that is, a list of the currently active variables together with their values. This utility provides a scrolled alphabetic listing of all variables in use including the contents of arrays of up to two dimensions. Other arrays are named, but the values of their elements are not given.

Use

The routine should be LOADed above or below the problem program, whichever is most convenient. The preferred method is to LOAD in 'List Variables' at the usual value of PAGE, and then set PAGE=PAGE+&500. Normal BASIC programs can now be LOADed and run.

If you require a symbol table, first place a STOP instruction at the point in your program where you would like to study the variables. When your program STOPS, reset PAGE to point to the utility and type GOTO 10 (*not* RUN !!) to produce your variable list. After the string, real and array variables have been given, you have the option of listing the resident integer variables – press Y to do so. Of course, if you only need to know the values of the variables at the end of your program, the STOP instruction is not needed.

When you return to your program to conduct further tests don't forget to reset PAGE. In fact, it may be possible to resume your program from just after the STOP, or even from an entirely different place provided you are not in a PROCedure, loop etc. However, it is unlikely that you would wish to use the utility in this way as it is somewhat risky.

If you intend to conduct a series of checks on your code, by far the best method is to include the coding for the utility with your program. To do so, amend it as follows:

```
 10 DEF PROCsymbol
 20 - delete
 30 LOCAL
 80 - delete
130 - delete
140 ENDPROC
```

and RENUMBER 30000. Save the new version as, say SYMB1.

The routine is now ready to be merged with your own program which should now be LOADed into store. Typing END <RETURN> will force the computer to calculate TOP – get it to PRINT TOP-2 for you. The merging is done by typing:

*LOAD "SYMB1" <The result of TOP-2>

This assumes that your program does not use line numbers of 30000 upwards, so best check that before you start. Once the programs have merged you can include as many PROCsymbol calls into your code as you like. Each will interrupt your program and print out the values of all variables used to date.

```
   10 REM LIST VARIABLES
   20 REMEMBER A%,B%
   30 !&70=A%:!&74=B%
   40 VDU 14
   50 PROCvar(&482,65):REM UPPER CASE
   60 PROCvar(&4C2,97):REM LOWER CASE
   70 *FX15,1
   80 B%=!&74
   90 PRINT'"Want A%-Z%?";
  100 VDU 7:A%=GET
  110 IF A%=89 PROCintegers
  120 PRINT
  130 A%=!&70
  140 END
  150
  160 DEFPROCvar(A%,B%)
  170 LOCAL X%,Y%,Z%
  180 FOR X%=0 TO 25
  190 Y%=?(2*X%+A%)+256*?(2*X%+A%+1)
  200 IF Y%<PAGE GOTO 330
  210 REPEAT
  220 Z%=?Y%+256*Y%?1
  230 Y%=Y%+1
  240 $&900=CHR$(B%+X%)
  250 Y%=Y%+1
  260 IF ?Y%=0 GOTO 290
  270 $&900=$&900+CHR$(?Y%)
  280 GOTO 250
  290 IF RIGHT$($&900,1)="(" THEN PROCar
ray:GOTO 310
  300 PRINT $&900;"=";EVAL($&900)
  310 Y%=Z%
320 UNTIL Y%<256
330 NEXT
340 ENDPROC
350
360 DEFPROCarray
370 LOCAL V%,W%
380 PRINT'"ARRAY: ";$&900;
390 PROCnamearray
400 W%=Y%?2+256*(Y%?3)
410 IF W%>9 W%=9
420 IF V%=4 PROConedim
430 IF V%=6 PROCtwodim
440 PRINT
450 ENDPROC
460
470 DEFPROCintegers
480 PRINT'"A%=";!&70
490 FOR A%=66 TO 90
500 PRINT CHR$A%;"%=";EVAL(CHR$A%+"%")
510 NEXT
520 ENDPROC
530
540 DEFPROCnamearray
550 LOCAL W%
560 FOR V%=2 TO Y%?1-1 STEP 2
570 W%=Y%?V%+256*(Y%?(V%+1))-1
580 PRINT STR$W%;",";
590 NEXT
600 VDU 127
610 PRINT")"
620 ENDPROC
630
640 DEFPROConedim
650 LOCAL V%
660 FOR V%=0 TO W%-1
670 $&A00=$&900+STR$V%+")"
680 PRINT $&A00;"=";EVAL($&A00)
690 NEXT
700 ENDPROC
710
720 DEFPROCtwodim
730 LOCAL T%,U%,V%
740 T%=Y%?4+256*(Y%?5)
750 IF T%>9 T%=9
```

```
760 FOR V%=0 TO W%-1:FOR U%=0 TO T%-1
770 $&A00=$&900+STR$V%+","+STR$U%+")"
780 PRINT $&A00;"=";EVAL($&A00)
790 NEXT,
800 ENDPROC
```

How it works

This is quite a complex utility which relies upon the fact that BASIC holds its variables in a special, well-organised way which we shall look at shortly. One complication is that, since the routine must print out your variables, it must not use any of its own! This is tricky but not impossible to achieve. Indeed an early version of this routine used absolutely no BASIC variables (how?) although it did suffer from the drawback that it was totally incomprehensible.

This version uses LOCAL integer variables (A% and B%,) which are stored before use and reloaded at the end, and the string indirection operator $ to generate its strings. The effect of using LOCAL variables T%–Z% is to preserve the values of these items once the PROCedure that uses them is complete. The illusion, having run the utility, is that no variables have been used at all.

To find a variable, we first look in Page 4. Here there is a table of pointers (that is addresses, stored low byte: high byte) to all the variables used by the current program:

&482/3	points to first variable starting with A
&484/5	points to first variable starting with B
&486/7	points to first variable starting with C
&487/8	. . .
. . .	. . .
&4B4/5	points to first variable starting with Z
&4C2/3	points to first variable starting with a
&4C3/4	. . .
. . .	. . .
&4F4/5	points to first variable starting with z

The variables themselves are stored at LOMEM (which is usually at the end of your program), in the order in which they are referenced by the program.

Integer variables @% to Z% are stored in the first part of Page 4, although we do not use that fact in this routine as it is easier to get at them using EVAL.

Let us move from the theoretical to the specific and look at a simple case which will help you understand this utility better.

Fig. 1 on page 52 shows a memory dump of the area assigned to a program after it has RUN. The three variables used, in the order in which they appear, are: base%, length%, and line_end%. (It is coincidental that these are in alphabetic order). Fig. 3 shows the relevant section of Page 4.

The entry at &4C4 points to the first variable starting with b and indicates &11D3. Now refer back to Fig. 1 and you will see that this is the value of TOP and the entry there (it's an address) is &0000. This is a pointer to the next variable beginning with b and, as there isn't one, the pointer is set to 0. Following the pointer is the rest of the name, ending with &00 (at &11D9) and then the actual value of the variable stored in the appropriate format.

```
0480  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0490  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04C0  00 00 00 00 D3 11 00 00 00 00 00 00 00 00 00 00
04D0  00 00 00 00 00 00 00 00 DE 11 00 00 00 00 00 00
04E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 3. Area of Page 4 memory showing pointers to variables

base% is clearly an integer and so it is stored in four bytes from &11DA to &11DD. Reals, strings and arrays each have their own formats but they need not concern us as we can use the powerful EVAL function to access their values without bothering to discover what they look like.

The only other entry (Fig. 3) in Page 4 is at &4D8 and points to the first variable beginning with l. It indicates &11DE which is assigned to the variable length%. This time, however, there is a link address to &11EB (Fig. 1) because there is another variable beginning with l, namely line_end%. The link address here is 0 indicating that there are no more variables beginning with l.

The utility works by starting at Page 4 and following the pointers until each variable has been found. When one is found, its name is built up at &900 (line 270) from where it can be EVALuated.

The process is a bit more complicated when an array is found although the principle is the same. See if you can work out the array part for yourself by studying the program listing. We will look at the method of storing string arrays – the most difficult type – when we introduced the string sort utility in the last chapter.

Alternatively, why not LOAD your memory display utility and use it to look at some simple programs that handle arrays? After a while the method of storage will become clear.

Without re-dimensioning the array it is possible to list its elements up to a subscript of 9 and this is done for arrays of one or two dimensions. Other arrays are named but the values of their elements are not given.

Procedures

PROCvar is the main module responsible for building each variable name and then EVALuating it. Two parameters are passed; the start address of the chain in Page 4, and the ASCII code of the opening character of the name.

If the variable turns out to be an array, which is detected by the rightmost character of the name being a ')', then PROCarray is called to deal with it. This names the array by calling PROCnamearray and prints out its elements, if appropriate, by calling PROConedim or PROCtwodim. PROCintegers is a simple procedure to print out the values of the resident integer variables.

Variables

None!!

Actually that's not strictly true: A% and B% are used once their original values have been stashed away ready to reload at the end of the routine. All other variables are necessarily resident integers, declared as being LOCAL to the PROCedures in which they feature.

Any strings that may be required are built up at &900, and possibly &A00, by means of the string indirection operator $. Thus the string variables used are $&900 and $&A00.

Extensions

Using similar coding to follow the links to the variable names, we could produce a 'cross-reference' listing (usually abbreviated to Xref) which shows by the name of each variable the references, by line number, to it. Xref's are used to track down the locations of PROCedure names and other labels, especially in long programs where a manual checking process would be unreliable. Unfortunately, to produce an Xref we would have to use a technique similar to that in 'Find' and then sort the list of names found, since Xref's should be in alphabetical order.

It is tempting to use the symbol table approach, as that will produce the names in order, but this cannot work because:

i) Only active variables will appear in the list. If a name has not yet been referenced it will not have an entry in the storage area.

ii) Having found a name, we would still have to scan the BASIC program to find the lines on which it occurred.

From these observations it would appear that Xref has more in common with 'Find' than it does with the symbol table and a true Xref would be quite a complex routine to produce.

One interesting exercise would be to produce a list of PROCedures together with the lines on which they are defined. This can be done by using the ideas mentioned here and your starting point is &4F6/7 which is where BASIC stores the reference to the first PROCedure. Your second, and final, clue is that you will need your memory display utility. Now it's up to you!

Finally, in the instructions for this utility it is suggested that you start it off with GOTO 10 as opposed to RUN. Why?

The diagnostic routines given in this section are among the most useful in the book. Like all tools, their effectiveness is greatly increased if you learn to use them properly, if possible by anticipating the circumstances in which you are likely to use them. For example, if you are in the middle of a delicate fault-finding situation, the last thing you want to be doing is LOADing in utilities, although sometimes it is unavoidable. Instead, plan ahead. If you have a troublesome program, LOAD in any required utilities first, so that they are available as soon as problems occur and, hopefully, before the situation becomes irretrievable. I have a utilities ROM chip fitted to my machine which includes a number of useful routines (many of them appear in this book) and it has saved me literally hours of work.

There are many steps to writing successful programs and planning and debugging are two very important aspects. The routines featured in the previous two sections should provide you with most of the debugging aids you are likely to need.

Section 3:

Sound and graphics

Although the BBC computer has only two BASIC commands for handling sound, this is one of the least understood areas of the machine. The difficulty is that both SOUND and ENVELOPE have complicated syntax which never seems to get any easier to use. Two utilities in this section are designed to ease the problem, but we don't get involved with some of the more specialised used of SOUND – for example, synchronisation.

Conversely, the problem with graphics is not so much the command syntax, as the bewildering array of commands. (In a sense, each of the PLOT commands is a different instruction and only a few are named – a good example is PLOT 5 which is equivalent to DRAW). The variety of available commands means that very sophisticated displays are possibly – if only you know how to get them!

Graphics utilities are very common (the most written BBC utility program of all is one of them) and programs relating to graphics can also be found in other sections of this book.

Incidentally, the fact that one of our graphics utilities is for MODE 7 should serve as a reminder that 'graphics' does not necessarily imply high-resolution.

Utility 17:

ENVELOPE editor

Description

The BBC computer can produce a wide range of sounds and effects through the use of its SOUND and ENVELOPE commands, although some effort is required to produce consistent, predictable results. In an attempt to provide us with as many sounds as possible, the designers have allowed these commands to expect four and fourteen parameters respectively and here is an immediate trade-off between flexibility and usability: the more things you can mess about with, the greater the range of effects – and the harder it is to get them. Thus, an obvious and essential utility for all potential sound users is one which allows you to change both SOUND and ENVELOPE parameters quickly and to try out the resulting sound with the minimum amount of fuss.

The routine presented here allows you to do just that and it will also display the BASIC SOUND and ENVELOPE statements that you have just built up. This is clearly necessary as they are the means by which the sound will eventually be played.

Anyone who is familiar with the scientific side of music or who has been involved with electronic instruments such as synthesisers will already be familiar with terms such as 'Attack' and 'Release', but unfortunately they are used in a different way by the computer. These are the times, normally measured in milli-seconds, occupied by certain sections of the amplitude envelope, but the software does not interpret them in this way and the relationship between their usual and Beeb specific meanings is extremely complicated.

If you have not come across these terms before, then you are entitled to be confused! A great many effects are possible and only a few of them can be adequately described in words – the best way to find out how the various parameters affect the sound is to try them for yourself and that is the purpose of the utility.

When you choose an envelope number the current state of that envelope is displayed along with a SOUND statement that will be used to play it. You may change the contents of any field by using the cursor keys to increase or decrease that parameter and at any time you can press P to play the sound. These facilities make it very easy to home-in on a sound, since it enables you to quickly adjust individual parameters and keep playing until you get the desired effect.

Use

There are no complications with this program and, as it is written entirely in BASIC, it can either be RUN or CHAINed.

After you have entered the number of the envelope that you wish to amend (the permitted range is 1–4 but this can be extended to include all 16 possible envelopes), the present state of the envelope is displayed in 14 fields (not quite the same as the 14 parameters) along with a SOUND statement that will be used when you want to play the sound. You can press one of the following keys at almost any time:

D: Display BASIC commands for both current ENVELOPE and SOUND. The ENVELOPE parameters are set up with the computer and will be retained even when you quit the utility, whereas the SOUND command is only relevant to this program and is not stored outside it.

F: Amend field. This is the most valuable facility of the program. Each of the displayed parameters has an associated field number which you enter following key F (then press <RETURN>). The cursor is moved to the correct field and you may at this point adjust the value of that parameter by using the cursor up/down keys.

N: New envelope. When you have finished programming one envelope this allows you to start on another. The SOUND parameters are reset to their starting values and the current state of the new envelope is displayed. Basically, all that happens is that the program restarts.

P: Play. The current SOUND/ENVELOPE combination is triggered so that you can listen to the sound you have just created.

Q: Quit the program. Returns you to BASIC and resets the cursor keys and auto-repeat speed.

S: Stop the sound. When you play a sound, it is quite easy to hang up a sound channel for longer than you intended. Pressing P several times will queue up requests to play the sound that are not serviced until the preceding requests are complete. S will send a higher-priority request to play a short burst of silence, effectively clearing the buffer. You will get advanced warning of a full buffer as the 'shift lock' LED will come on.

Cursor up/down: When the cursor has been moved to a field following F these keys increase or decrease the value in that field. The keys auto-repeat and they have been speeded up so that you can rapidly reach any required value. One of the problems with the envelope command is that each parameter seems to have its own range of values ('seems' is right – with one exception, all fields are 0–255 as they are held in one byte, but we try to stay consistent with the manual) and validating the data can be difficult. With this utility there are no such problems; when you reach either end of the scale for the field you are amending, it 'wraps-round' and takes you back to the other end of the scale without stopping. This is very convenient because it means that you don't even need to worry about what the numbers are – they are always 'correct' so your only problem is getting the sound right.

Suppose you start with an empty machine and select ENVELOPE 1. If you choose to amend field 13 (ALA) by pressing 'cursor up' it will quickly run through values from 0 upwards. However, if you press 'cursor down', it will start from 0, go (down!) to 126 – its greatest possible value – then run down from 126 until it reached 0, when the cycle would be repeated. This wrap-around effect applies to all fields including field 1 (Auto repeat pitch envelope) which can only take two values: 0 (no auto-repeat) and 1 (auto-repeat). You may have noticed that this is the opposite setting from bit 7 of the second parameter of the ENVELOPE command as described in the *User Guide*. This is not an error – it seems more sensible that '1' should mean 'Yes – do auto-repeat' while 0 means 'No, don't bother'. Change it if you don't agree. The only field you cannot change is the envelope number in the SOUND command – the reason for this should be obvious.

If you intend to use the ENVELOPE command at all you need a utility such as this one to help you produce the desired results. The effects of the various parameters on the overall sound are almost impossible to predict accurately without much experience or prior calculation. The purpose of this program is to remove much of the guesswork from this otherwise difficult task.

```
 10 REM ENVELOPE EDITOR
 20 *FX 4,1
 30 *FX12,4
 40 @%=4
 50 REPEAT
 60 REPEAT
 70 MODE 7
 80 INPUT"Envelope number? "env%
 90 UNTIL env%>0 AND env%<5
100 PROCinitial
110 PROCscreen
120 PROCfillfields
130 REPEAT
140 F%=INKEY(0)
150 IF F%=68  PROCdisplay
```

```
 160 IF F%=70  PROCfield
 170 IF F%=80  PROCplay
 180 IF F%=83  PROCstop
 190 IF F%=138 PROCdec(field%)
 200 IF F%=139 PROCinc(field%)
 210 UNTIL F%=78 OR F%=81
 220 UNTIL F%=81
 230 PRINTTAB(0,23)
 240 *FX 4,0
 250 *FX12,0
 260 END
 270
 280 DEFPROCinitial
 290 base%=&8AE+16*env%
 300 field%=0:min%=-255:max%=255
 310 chan%=1:pitc%=100:durn%=100
 320 ENDPROC
 330
 340 DEFPROCscreen
 350 CLS
 360 PRINT"ENVELOPE ";env%;" PARAMETER
S:"''
 370 PRINT" 1Auto repeat pitch envelo
pe"
 380 PRINT" 2Step length ( x 0.01 sec
s)"'
 390 PRINT" 3PI1"SPC(13)"9AA"
 400 PRINT" 4PI2"SPC(12)"10AD"
 410 PRINT" 5PI3"SPC(12)"11AS"
 420 PRINT" 6PN1"SPC(12)"12AR"
 430 PRINT" 7PN2"SPC(12)"13ALA"
 440 PRINT" 8PN3"SPC(12)"14ALD"''
 450 PRINT"SOUND PARAMETERS:"'
 460 PRINT"15Channel"SPC(8)"16Pitc
h"
 470 PRINT"  Envelope       17Dura
tion"
 480 ENDPROC
 490
 500 DEFPROCfillfields
 510 FOR I%=1 TO 18
 520 PROCcurs(I%)
 530 PRINT V%
 540 NEXT
```

```
  550 ENDPROC
  560
  570 DEFPROCcurs(Z%)
  580 PROCread(Z%)
  590 IF Z%=1  V%=1+((base%?2)>127)
  600 IF Z%=2  V%=(base%?2) AND 127
  610 IF Z%=15 V%=chan%
  620 IF Z%=16 V%=pitc%
  630 IF Z%=17 V%=durn%
  640 IF Z%=18 V%=env%
  650 IF V%>min%+255 V%=V%-256
  660 PRINT TAB(xtab%,ytab%);
  670 ENDPROC
  680
  690 DEFPROCread(Z%)
  700 RESTORE
  710 FOR J%=1 TO Z%
  720 READ xtab%,ytab%,min%,max%
  730 NEXT
  740 V%=base%?Z%
  750 ENDPROC
  760
  770 DEFPROCdisplay
  780 VDU 28,0,24,39,20,12
  790 PRINT "ENVELOPE ";env%;
  800 FOR I%=2 TO 14
  810 PROCread(I%)
  820 IF V%>max% V%=V%-256
  830 PRINT ",";V%;
  840 NEXT
  850 PRINT''"SOUND ";chan%;",";env%;","
;pitc%;",";durn%
  860 VDU 26
  870 ENDPROC
  880
  890 DEFPROCfield
  900 REPEAT
  910 VDU 28,0,24,39,20,12
  920 INPUT TAB(0,1)"Field? "field%
  930 UNTIL field%>0 AND field%<18
  940 VDU 12,26
  950 PROCcurs(field%)
  960 ENDPROC
  970
```

```
 980 DEFPROCplay
 990 SOUND chan%,env%,pitc%,durn%
1000 ENDPROC
1010
1020 DEFPROCstop
1030 SOUND 16+chan%,0,0,1
1040 ENDPROC
1050
1060 DEFPROCinc(Z%)
1070 IF Z%=0 ENDPROC
1080 V%=V%+1
1090 IF V%>max% V%=min%
1100 PROCupdate
1110 ENDPROC
1120
1130 DEFPROCdec(Z%)
1140 IF Z%=0 ENDPROC
1150 V%=V%-1
1160 IF V%<min% V%=max%
1170 PROCupdate
1180 ENDPROC
1190
1200 DEFPROCupdate
1210 PRINTTAB(xtab%,ytab%)V%;
1220 IF Z%=1  base%?2=base%?2 EOR 128
1230 IF Z%=2  V%=V%+((base%?2)AND 128)
1240 IF Z%=15 chan%=V%
1250 IF Z%=16 pitc%=V%
1260 IF Z%=17 durn%=V%
1270 IF Z%>1 AND Z%<15 base%?Z%=V%
1280 ENDPROC
1290
1300 DATA 32,3,0,1,32,4,0,127,9,6,-128,
127,9,7,-128,127
1310 DATA 9,8,-128,127,9,9,0,255,9,10,0
,255,9,11,0,255
1320 DATA 29,6,-127,127,29,7,-127,127,2
9,8,-127,0,29,9,-127,0
1330 DATA 29,10,0,126,29,11,0,126,13,16
,0,3
1340 DATA 33,16,0,255,33,17,-1,254,13,1
7,0,16
```

How it works

The data for the four regular envelopes is stored at the end of Page 8 (the data for ENVELOPE 1 starts at &8C0; each envelope having 16 bytes to itself) and the parameters are read from, and written to, this area. As soon as the ENVELOPE number has been selected, the SOUND parameters are reset and the base address for the ENVELOPE information is calculated. When the screen is displayed, the current ENVELOPE information is filled in and the program enters a loop and waits for you to press one of the relevant keys – each key has an associated PROCedure to carry out its processing.

To add some colour to the display, the original program contained colour control codes in some of the PRINT statements and, of course, these cannot be reproduced in the printed listing. Where a colon (:) appears in a PRINT statement in lines 360–470 and the INPUT command in line 920, it should be replaced by a suitable colour code (obtained by pressing SHIFT and a function key simultaneously as explained in the *User Guide*). The colours used – which are not necessarily the most tasteful – were:

Line numbers	Colours
360,450,920	Cyan
370,380	Red, green, yellow
390–440,460	Red, green, yellow, red, green, yellow
470	Green, yellow, red, green, yellow

You can, of course, replace these codes with spaces and settle for an all-white display, but this can be difficult to read as it contains so much information.

At the end of the program is the DATA for each of the eighteen fields. Each field has four entries and these are (in order): horizontal and vertical screen co-ordinates of the field, then minimum and maximum allowable values for that field. Each time a field is selected by pressing F this information is read and used to update that field until a new field is chosen.

Procedures

To restore the starting values for a new, or the first, envelope PROCinitial is called followed by PROCscreen to build the display. The screen so created is only a skeleton containing the fixed data – the rest of the screen is completed by PROCfillfields which does just that by reading the relevant information from the Page 8 entry for the envelope and then converting it to a form suitable for display.

Each of the above PROCedures is only required during the initialisation process for a particular envelope and is not needed thereafter.

The remaining PROCedures may be called at various stages of the processing.

One of the most important modules is PROCcurs which takes as its sole parameter a field number in the range 1–18. The function of this routine is to move the cursor to the correct place on the screen and read the current value of that field into the variable v% (the v stands for 'value'). To help this PROCedure read the data that describes the fields, PROCread is called, again with the field number as its parameter. Since the value of most of the fields is simply held at a displacement from the base of the data for each envelope, this PROCedure can make a fairly good attempt at reading the value; if it subsequently turns out to be wrong, PROCcurs will correct it. (If this sounds a bit hit-and-miss see note on base% later).

Although the information on the screen is fairly close to what BASIC would expect, it is useful to see the current ENVELOPE and SOUND statements displayed with their correct BASIC syntax. Pressing D will display the commands by calling PROCdisplay to set up a text window and then to read the bytes from their Page 8 locations so that they may be displayed on the screen. Once again the DATA statements are used to ensure that each field is in exactly the right range as given in the *User Guide*.

The most common activity in this utility is altering fields and PROCfield is called each time you press F. This PROCedure vets the field number to ensure it's in the range 1–18 and then calls PROCcurs to move the cursor and to read the minimum and maximum allowable values. The actual changes are done by PROCinc and PROCdec which are summoned in response to the cursor up and down keys respectively. These PROCedures update both the screen and Page 8 entries for the field that is used as the parameter for the PROCedure.

PROCplay and PROCstop are simple one-liners which start and stop the sound. PROCplay can be pressed anytime and enables you to hear the sound you have just established.

Finally, PROCupdate is called when the value of a field is altered – it is used to update (what else?) the screen and Page 8 entries for the envelope. Because of the way the program has been written, there is no reason why you cannot update fields while the sound is playing, especially if it has quite a long decay period. This is very useful as it means that tricky adjustments to the sound can be carried out on the spot and it speeds up the process of designing both SOUND and ENVELOPE commands.

Variables

The current envelope number is stored as env% which retains this value until you change envelopes with the N key. Similarly, at any time only one field will be current and its number is stored in field% – this can only be changed by pressing F to enter a new field number. Once the envelope number has been selected, the data associated with that envelope can be located in Page 8 and base% is used as a base for this data – notice that it is actually located two bytes before the true start of the data. This is very convenient for the program as the byte for field n will then be located at (base% + n) where n is in the range 3–14.

We have already seen that each field has four constants associated with it and stored in the DATA statements at the end of the program. The first two items are xtab% and ytab% which are used by the screen-handling PROCedure to position the cursor in the right place when the field is chosen. The remaining items are min% and max% and these define the limits that the field can reach. If you attempt to select a value outside one of these limits, then the value will be reset to the other limit as though nothing had happened. We have used this concept in several programs, where it has been called 'wrap-around', especially when used in the context of screen movement, and the effect in this program is close enough to merit the same name.

The contents of the fields associated with the ENVELOPE command are not held in any variables: instead they are read directly from the envelope area in Page 8. However the parameters for the SOUND command are held as program variables and channel, pitch and duration are referred to as chan%, pitc% and durn% respectively.

Throughout the program, V% contains the value of a field and Z% is a parameter passed into a PROCedure.

Extensions

The computer will recognise ENVELOPE numbers up to 16 and the storage for those beyond 4 will overflow into Page 9. Provided you do not intend to use this area – it is used by BPUT and SPOOL, and by some of our utilities – then you can amend line 90 so that envelope numbers up to 16 are acceptable.

Once you have set up some envelopes, you can display the relevant BASIC statement and then write it down but this can be tedious, especially if you have to copy 16 such statements off the screen. An obvious amendment would be to include a save facility that *SAVEd the contents of the envelope areas so that they could be read back and used by other programs. These programs would not need to execute ENVELOPE statements as that is equivalent to setting up the data in Pages 8 and 9 and this would be done by a *LOAD.

Some envelope editing programs actually depict the pitch and amplitude contours as a function of time using a high resolution MODE to display the graph. This looks impressive and does convey something of what the sound will turn out like. Unfortunately, unless you have some experience of this sort of graph, it may be found to be rather too confusing. Furthermore, any particularly subtle effects would probably not show up on the graph (though, to be fair, they would not be obvious from the numeric data either).

However, because the routine is so short there is plenty of room to add a graphics facility if required and anyway it would be an interesting extension to program. One feature that we have already noted will cause a few problems and that is that none of the parameters is actually measured in time units. Instead, those parameters feature as 'gradients' (or amount per step of time), and the conversion between the two ways of representing the data is very complicated. A worthwhile addition to the routine is to allow the attack, decay, sustain and release times to be entered in centi-seconds and then to compute the corresponding ENVELOPE and SOUND parameters. Until this is done, you can hardly consider drawing a graph of the resulting ENVELOPE contour.

Utility 18:

Music processor

Description

In the previous utility, we introduced a method of programming the ENVELOPE command so that your computer could produce a wide range of sound effects. We now turn to the problem of providing data for those envelopes to work from. One common use of sound is to provide background effects for games and, when used in this way, the musical content is unimportant – humour and variety are usually the order of the day. However, there is a more serious side to the use of the music on the computer and in this the SOUND command takes on a more important role.

'Music' suggests discipline and form and persuading your computer to play anything but the simplest tunes is surprisingly difficult. In fact, it not so much the commands that cause the problems but finding data with which to supply them. Like many of the most complex situations that computers have to deal with, the solution to this one comes down to data organisation.

Assuming that you are using the machine to play some already existing tune, you obviously have to tell the micro what that tune is, and as your computer cannot read music, there is already a communication problem. A possible solution is to tell it what notes to play, and how long to play them for, by means of a DATA statement. For example:

```
10 REPEAT
20 READ pitch,length
30 SOUND 1,-15,pitch,length
40 UNTIL length=0
50 DATA 61,10,73,20,81,10,89,20,93,4
60 DATA 89,6,81,20,69,10,53,20,0,0
```

There are, however, a number of drawbacks to this simple approach. The above code plays the first line of a famous tune but you would be very hard pushed to guess what it was just by looking at the listing. Editing or extending the tune is only possible through a laborious process of converting note values into numeric data and then packing them into DATA statements.

Another approach that is sometimes used is to store the information in strings, say one per bar or line of the tune, and then to extract it at play(ing)-time by using the command MID$. The overall results obtainable by these methods are adequate – we won't attempt to better them – but the process is both time-consuming and error prone.

This utility allows you to enter music data in a more meaningful way so that it may be reviewed or edited quickly and easily. Once entered, the data is converted to a special format used by this program and the 'duet' routine given later. It is then stored ready to be recalled later. Although the main purpose of the routine is to enable you to set up music data for use in other programs, the utility is fun to use and, if your music theory is a bit dodgy, you will also find it quite instructive. At no time will you see notes represented by numbers (although they are, really) – they will appear in standard musical notation or as keys on a keyboard.

The next section explains what facilities are available and how they are used but first, here is a summary of the program and its associated routines.

Data is entered one bar at a time by using the keyboard as a 'piano' to play the notes. As notes are entered they are placed in their correct positions in the bar, which is displayed graphically, as is the relationship between the piano and QWERTY keyboards. When the bar is valid (musically, as opposed to aesthetically) it can be stored and you proceed to the next bar. At any point you may edit a previously saved bar, store the whole tune on tape or play it back without affecting the data you have stored. When you break off like this you can return to the tune and type in more information until it is finished. Finally, when the tune is complete you should SAVE it on tape or disc.

Once you have stored the data it may be reloaded at a later stage by this routine, although you will probably want to use the tune in your own program and we show you how to do this. To replay the music from within your program only requires a short routine which can be copied from the main utility, although the exact format depends on just what you want to do with the tune.

The method of storage allows each note the characteristics of pitch and length while each bar has parameters which specify an ENVELOPE and a register (octave) to be used for the duration of the bar. These features can make up for the rather monotonous nature of computer music by introducing variations into the sound.

I would like to thank John Rawcliffe for his help with the development of this program.

Use

This is a very long program and it uses MODE 1 graphics, which occupy most of the memory. To compensate for this we have split the processing into two parts. The first part defines ENVELOPEs and some characters using the VDU 23 command. Once this has been done, the code is no longer needed and so it can be overwritten by the main program. Whether you use tape (PAGE=&E00) or discs (PAGE=&1900), you only have to run the 'Header' program which sets PAGE to &1300 (which is appropriate to both tape and disc) and this then chains the main program.

When you first enter the program or after you have completed the task you are doing, you will be confronted by the initial option screen. This requires you to press a single key and the options are as follows:

A: Append data to the existing tune. You can only do this if there is an existing tune, and you have not used up your 100 bar allowance.

C: Create tune. The previous tune is forgotten and you must enter the time signature (either 3/4 or 4/4) for the new tune.

E: Edit existing tune. The unit of editing is a bar and you must enter the number of the bar you wish to edit. You can do what you like to the bar, but you can only enter the bar when the time values of the notes and rests add up to the time signature. This validation is applied to every bar you enter.

L: Load data from tape or disc. After you have given the title, the data in memory is overwritten from the data on the file. This data will previously have been saved by the utility.

P: Play the tune – a reasonably useful facility! Select the tempo from 1 (extremely slow) to 20 (extremely fast) and the tune will be played with each bar number being displayed on the screen as the bar is playing.

S: Save data to tape or disc. Your tune is stored ready to be used by this, or your own program.

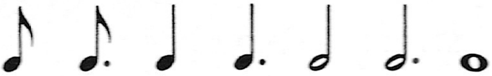
X: Exit the program.

Three of these facilities take you on to the main screen which consists of a status line, a bar's worth of stave, and a 'keyboard' layout – at this point, quite a number of keys become active. All of the following also applies to editing, but that uses two additional keys so let us concentrate firstly on creating and appending a single bar.

Whenever you press one of the 'keyboard' keys, the corresponding note is played, taking into account the register (called 'octave shift'), ENVELOPE, and note length. The note is also added to the bar using standard musical notation. When the bar is full, no further notes can be added (neither will they sound) and you must press

<RETURN> to enter that bar and proceed to the next one. In the case of 'Edit', you are returned to the main screen.

S cycles the octave shift through the values 0 (low), 2 (middle) and 4 (high) and key V cycles the ENVELOPE numbers through 1–4. When the full bar is entered, the current, or latest, values of these fields are stored with the bar and will be used later when the bar is played. Key L is used to vary the length of the note that will be entered when you press a 'note' key. The range of values is:



To enter a rest of the same value as the current note length, press the space bar; the rest is indicated by placing the note on the middle line and writing an 'R' over it. The effect of the <DELETE> key is to remove the previous note from the bar and to re-position the cursor ready to rewrite it.

Finally, to escape from this mode back to the option screen, press X when you are at the start of an empty bar. It is not possible to exit unless you have tidied up all the loose ends.

When you are in the 'Edit' mode, <DELETE> removes the current note – the one indicated by the edit arrow – and the cursor is not moved. To step the cursor left and right, use the < and > keys respectively; the movement wraps-around the bar. The only significant differences between 'Edit' mode and the others are that you have to move the cursor yourself and that you can enter an 'untidy' bar provided the note lengths all add up properly.

This covers the use of the routine, but you will want to know how to retrieve the data yourself so that you can play the music in your own program. Some details on how to do this are given in the Extensions section below.

```
   10 REM MUSIC HEADER
   20 ENVELOPE 1,3,0,0,0,0,0,0,126,-10,-
5,-2,120,60
   30 ENVELOPE 2,2,0,0,-1,2,2,1,120,-10,
-5,-10,120,80
   40 ENVELOPE 3,132,8,-8,0,1,1,0,60,-20
,0,-20,120,30
   50 ENVELOPE 4,5,1,-1,1,1,1,1,30,-8,0,
-20,126,101
   60 VDU 23,224,-1,-1,-1,-1,-1,-1,-1,-1
   70 VDU 23,225,-2,-2,-2,-2,-2,-2,-2,-2
   80 VDU 23,226,-8,-8,-8,-8,-8,-8,-8,-8
   90 VDU 23,227,31,31,31,31,31,31,31,31
  100 VDU 23,228,12,14,15,13,12,12,12,12
  110 VDU 23,229,12,12,12,12,12,12,12,12
  120 VDU 23,230,60,124,-4,-4,-4,-4,124,
60
  130 VDU 23,231,60,124,204,204,204,204,
124,60
  140 VDU 23,232,60,126,225,193,131,135,
126,60
  150 VDU 23,233,60,60,60,60,0,0,0,0
  160 VDU 23,234,-8,-4,6,3,3,3,6,60
  170 VDU 23,235,12,28,60,60,108,108,204
,204
  180 VDU 23,236,60,6,3,3,3,6,-4,-8
  190 VDU 23,237,204,204,-1,-1,12,12,12,
12
  200
  210 PAGE=&1300:CHAIN "MUSIC2": REM HOP
EFULLY THIS FILE ("MUSIC2") CONTAINS THE
 PROGRAM GIVEN BELOW
```

```
   10 MODE 1
   20 DIM table 1010,temp 8,len(6)
   30 *FX 4,1
   40 PROCinit
   50 PROCsetup
   60 REPEAT
   70 VDU 4,26,12
   80 PRINT'"Your choice: (A,C,E,L,P,S,X
):";
   90 PROCvet("ACELPSX"):PRINT G$:f%=F%
  100 IF f%=1 AND nobars%>1 PROCappend
  110 IF f%=2 PROCcreate
  120 IF f%=3 AND nobars%>1 PROCedit
  130 IF f%=4 PROCload
  140 IF f%=5 AND nobars%>1 PROCplay
  150 IF f%=6 AND nobars%>1 PROCsave
  160 UNTIL f%=7
  170 *FX 4
  180 MODE 7
  190 END
  200
  210 DEFPROCinit
  220 note$="":r1$=""
  230 VDU 19,1,5,0,0,0,19,2,5,0,0,0
  240 line$="1100000000000000000007770"
  250 shrp$=".#.#..#.#.#..#.#..#.#.#.."
  260 tpos$="AABBCDDEEFFGHHIIJKKLLMMNG"
  270 keys$="Q2W3ER5T6Y7UI9O0P@^[\_"+CHR
$136
  280 keys$=keys$+CHR$139+" "+CHR$127+CH
R$13+"LVX,.S"
  290 FOR I%=1 TO 16:READ K%:note$=note$
+CHR$K%:NEXT
  300 DATA 228,10,8,230,229,10,8,230,229
,10,8,231,32,10,8,232
  310 FOR I%=0 TO 6:READ len(I%):NEXT
  320 DATA .5,.75,1,1.5,2,3,4
  330 FOR I%=1 TO 14:READ K%:r1$=r1$+CHR
$(223+K%):NEXT
  340 DATA 1,3,4,3,4,2,1,3,4,3,4,3,4,2
  350 r1$=r1$+r1$
  360 r2$=STRING$(14,CHR$224+CHR$225)
  370 ENDPROC
  380
  390 DEFPROCcreate
  400 PRINT'"Time (3)/4 or (4)/4 ?";
  410 PROCvet("34"):PRINT G$:sig=F%+2
  420 FOR I%=0 TO 996 STEP 4:table!I%=0:
NEXT
  430 PROCsetup
  440 PROCappend
  450 ENDPROC
  460
  470 DEFPROCsetup
  480 nobars%=1:s%=2:env%=1:tempo=13:oct
%=0
  490 ENDPROC
  500
  510 DEFPROCappend
  520 bar%=nobars%
  530 PROCkeys
  540 REPEAT
  550 PROCvet(keys$)
  560 IF F%<26 PROCnewnote
  570 IF F%=26 AND ptr%>1 PROCdelete
  580 IF F%=27 AND FNfull PROCbarfull
  590 IF F%=28 PROCnotelength(1)
  600 IF F%=29 PROCenv(1)
  610 IF F%=33 PROCoct(1)
  620 UNTIL F%=30 AND ptr%=1
  630 table?(bar%*10)=255
  640 ENDPROC
  650
  660 DEFPROCkeys
  670 VDU 26,12,4,23,1,0;0;0;0;
  680 PRINT " BARS   BAR    ENVELOPE   N
OTE   OCTAVE"
  690 PRINT " FREE  NUMBER   NUMBER   LE
NGTH  SHIFT"
  700 PRINT TAB(0,22)
  710 FOR I%=1 TO 3:PRINT TAB(6)r1$:NEXT
  720 FOR I%=1 TO 4:PRINT TAB(6)r2$:NEXT
  730 VDU 5:GCOL 0,1
  740 MOVE 240,315:PRINT"2 3   5 6 7   9
 0   ^ \"
  750 MOVE 204,130:PRINT"Q W E R T Y U I
 O P @ [ `"
  760 MOVE 50,580:VDU 235
  770 MOVE 50,548:VDU 237
  780 MOVE 50,680:VDU 231+sig
  790 MOVE 50,648:VDU 233+sig
  800 PROCstave
  810 ENDPROC
  820
  830 DEFPROCstave
  840 ptr%=1
  850 VDU 4,28,3,20,39,6,12,26
  860 GCOL 0,3
  870 FOR I%=500 TO 692 STEP 48
  880 MOVE 100,I%:DRAW 1200,I%:NEXT
  890 MOVE 100,500:DRAW 100,692
  900 MOVE 1200,500:DRAW 1200,692
  910 PRINT TAB(2,5);101-nobars%;" "
  920 PRINT TAB(9,5);bar%
  930 PROCenv(0)
  940 PROCoct(0)
  950 PROCnotelength(0)
  960 ENDPROC
  970
  980 DEFPROCnewnote
 990 IF FNfull ENDPROC
1000 IF len(s%)+tot>sig ENDPROC
1010 IF nobars%<100 PROCupdate
1020 ENDPROC
1030
1040 DEFPROCdelete
1050 ptr%=ptr%-2^D%
1060 L%=FNnote(ptr%) DIV 32
1070 P%=FNnote(ptr%) MOD 32
1080 PROCdeletenote
1090 K%=ptr%
1100 REPEAT:K%=K%-1
1110 UNTIL FNnote(K%)
1120 D%=FNnote(K%) DIV 64
1130 ENDPROC
1140
1150 DEFPROCbarfull
1160 table?(bar%*10-1)=oct%
1170 table?(bar%*10)=env%
1180 bar%=bar%+1
1190 nobars%=nobars%+1
1200 PROCstave
1210 ENDPROC
1220
1230 DEFPROCnotelength(Z%)
1240 VDU 4,23,1,0;0;0;0;
1250 IF Z% s%=(s%+1) MOD 7
1260 K%=s% DIV 2
1270 PRINT TAB(28,4)MID$(note$,4*K%+1,4
)
1280 IF s% AND 1 K%=233 ELSE K%=32
1290 PRINT TAB(29,5)CHR$K%
1300 ENDPROC
1310
1320 DEFPROCenv(Z%)
1330 VDU 4,23,1,0;0;0;0;
1340 IF Z% env%=env% MOD 4 +1
1350 PRINT TAB(18,5);env%
1360 ENDPROC
1370
1380 DEFPROCoct(Z%)
1390 VDU 4,23,1,0;0;0;0;
1400 IF Z% oct%=(oct%+2) MOD 6
1410 PRINT TAB(35,5);oct%
```

```
 1420 ENDPROC
 1430
 1440 DEFPROCupdate
 1450 P%=F%:L%=s%
 1460 PROCdraw
 1470 PROCpn
 1480 table?((bar%-1)*10+ptr%)=32*L%+P%
 1490 ptr%=ptr%+2^D%
 1500 ENDPROC
 1510
 1520 DEFPROCplay
 1530 REPEAT
 1540 REPEAT
 1550 CLS
 1560 INPUT'"Tempo: 1=Slow, 20=Fast "tem
po
 1570 UNTIL tempo>0 AND tempo<21
 1580 tempo=22-tempo
 1590 FOR bar%=1 TO nobars%-1
 1600 oct%=FNnote(9):env%=FNnote(10)
 1610 PRINT TAB(0,9)"Playing bar ";bar%;
 1620 FOR I%=1 TO 2*sig
 1630 T%=TIME
 1640 L%=FNnote(I%) DIV 32
 1650 P%=FNnote(I%) MOD 32
 1660 IF P% PROCpn
 1670 REPEAT UNTIL TIME>T%+tempo
 1680 NEXT I%,bar%
 1690 PRINT''"Play it again, (Sam)? Y/N
";
 1700 PROCvet("YN")
 1710 UNTIL F%=2
 1720 ENDPROC
 1730
 1740 DEFPROCpn
 1750 IF P%=25 V%=0 ELSE V%=env%
 1760 SOUND 1,V%,24*oct%+4*P%,len(L%)*te
mpo
 1770 ENDPROC
 1780
 1790 DEFPROCedit
 1800 REPEAT
 1810 PRINT'"Edit which bar";
 1820 INPUT bar%
```

```
1830 UNTIL bar%>0 AND bar%<nobars%
1840 oct%=FNnote(9):env%=FNnote(10)
1850 PROCkeys
1860 FOR I%=1 TO 8
1870 L%=FNnote(I%) DIV 32
1880 P%=FNnote(I%) MOD 32
1890 IF P% PROCdraw:ptr%=ptr%+2^D%
1900 NEXT
1910 ptr%=1
1920 PROCarrow
1930 REPEAT
1940 L%=FNnote(ptr%) DIV 32
1950 P%=FNnote(ptr%) MOD 32
1960 PROCvet(keys$)
1970 IF F%<26 PROCinsert
1980 IF F%=28 PROCnotelength(1)
1990 IF F%=29 PROCenv(1)
2000 IF F%=31 PROCcursor(-2)
2010 IF F%=32 PROCcursor(0)
2020 IF F%=33 PROCoct(1)
2030 IF F%=26 AND P% PROCdeletenote
2040 UNTIL F%=27 AND FNfull
2050 C%=1:temp!1=0:temp!5=0
2060 FOR I%=1 TO 8
2070 K%=FNnote(I%) DIV 64
2080 P%=FNnote(I%) MOD 32
2090 IF P% temp?C%=FNnote(I%):C%=C%+2^K
%
2100 NEXT
2110 FOR I%=1 TO 8
2120 table?((bar%-1)*10+I%)=temp?I%
2130 NEXT
2140 table?(bar%*10-1)=oct%
2150 table?(bar%*10)=env%
2160 ENDPROC
2170
2180 DEFPROCcursor(K%)
2190 PROCarrow
2200 ptr%=(ptr%+K%+8) MOD 8 +1
2210 PROCarrow
2220 ENDPROC
2230
2240 DEFPROCarrow
2250 GCOL 3,3
```

```
2260 MOVE 100+120*ptr%,400
2270 VDU 5,94
2280 ENDPROC
2290
2300 DEFPROCinsert
2310 IF P% PROCdraw
2320 PROCupdate
2330 ptr%=ptr%-2^D%
2340 ENDPROC
2350
2360 DEFPROCdeletenote
2370 PROCdraw
2380 table?((bar%-1)*10+ptr%)=0
2390 ENDPROC
2400
2410 DEFPROCsave
2420 PROCtitle
2430 X=OPENOUT(title$)
2440 BPUT# X,sig:BPUT# X,nobars%
2450 REPEAT
2460 V%=table?K%:BPUT# X,V%:K%=K%+1
2470 UNTIL V%=255
2480 CLOSE# X
2490 ENDPROC
2500
2510 DEFPROCload
2520 PROCtitle
2530 X=OPENUP(title$)
2540 sig=BGET# X:nobars%=BGET# X
2550 REPEAT
2560 V%=BGET# X:table?K%=V%:K%=K%+1
2570 UNTIL V%=255
2580 CLOSE# X
2590 ENDPROC
2600
2610 DEFPROCtitle
2620 K%=1
2630 REPEAT
2640 PRINT'"Title (max 7 characters) ";
2650 INPUT title$
2660 UNTIL LENtitle$>0 AND LENtitle$<8
2670 ENDPROC
2680
2690 DEFPROCvet(A$)
```

```
 2700 REPEAT
 2710 *FX 21,0
 2720 G$=GET$:F%=INSTR(A$,G$)
 2730 UNTIL F%>0
 2740 ENDPROC
 2750
 2760 DEFPROCdraw
 2770 VDU 5:GCOL 3,3
 2780 dx%=120*ptr%
 2790 dy%=24*(ASCMID$(tpos$,P%,1)-64)
 2800 ln%=48*VALMID$(line$,P%,1)
 2810 IF ln% MOVE 90+dx%,400+ln%:DRAW 14
0+dx%,400+ln%
 2820 GCOL 3,1
 2830 IF MID$(shrp$,P%,1)="#" MOVE 60+dx
%,440+dy%:PRINT"#"
 2840 IF (L% AND 1) MOVE 140+dx%,445+dy%
:VDU 233
 2850 IF P%=25 MOVE 100+dx%,575+dy%:PRIN
T"R"
 2860 MOVE 100+dx%,472+dy%
 2870 D%=L% DIV 2
 2880 PRINT MID$(note$,4*D%+1,4)
 2890 ENDPROC
 2900
 2910 DEFFNfull
 2920 tot=0
 2930 FOR I%=1 TO 8
 2940 L%=FNnote(I%) DIV 32
 2950 P%=FNnote(I%) MOD 32
 2960 IF P% tot=tot+len(L%)
 2970 NEXT
 2980 =(tot=sig)
 2990
 3000 DEFFNnote(Z%)=table?(10*bar%-10+Z%
)
```

How it works

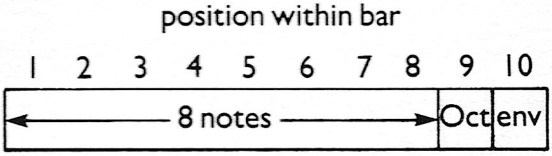
Once you have chosen a facility from the option screen, control is passed to one of the PROCedures described in the next section. It is at the next level down that the real flow of the program is established and, although each PROCedure is quite simple in itself, the interaction between them is complicated and you should study the program listing carefully, especially if you intend to amend the routine. At this level, the most important feature is the way that the data is stored – in fact, the whole of the program is affected by the format of this data. In many respects, the most important decision when designing a program such as this is 'How should the data be represented within the computer? A good choice will simplify much of the work that follows.

The contents of the bars are stored in a table 1010 bytes long, and each bar occupies exactly 10 bytes. The data is terminated by a 'bar' whose last byte is &FF (decimal 255) and so there is room for 100 bars of data in the table. Because the bars are a fixed length, it is easy to calculate the start address of any one by using the formula:

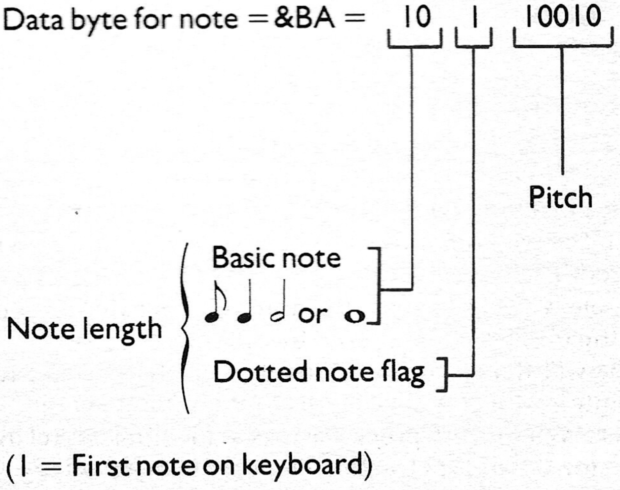
start address = table + 10 * bar number – 9

and this is useful for editing the bars.

Each bar can contain a maximum of eight notes and these occupy the first eight locations in the bar; the remaining two bytes contain the octave shift and envelope numbers to be used for playing that bar. This is shown in the diagram below:


position within bar
1 2 3 4 5 6 7 8 9 10
8 notes
Oct
env

The data for each note is extracted from the byte for that note like this:


Data byte for note = &BA = 10 1 10010
Pitch
Note length
Basic note
or
Dotted note flag
(1 = First note on keyboard)

Thus this note is F in the second octave and its length is a dotted minim. We have seen that a setting of &FF is used to indicate the end of the tune and two other special settings are used. These are: 0, which represents no note (this is not the same as a rest) and a pitch value of 25 which means that the note is a rest, with the length being indicated in the usual way.

When you SAVE the tune to your filing system, it is this table that gets stored and if you want to write your own playing routine, it will first need to read the table back in again. The data has been designed in this way so that editing is easy– it is possible to pack the data before it is stored and expand it again as you reload it; unless you have some special reason for doing so it is probably not worth the effort involved.

Procedures

The list below gives the cast (in order of appearance) and explains the function of each PROCedure.

PROCinit is called once to set up some strings and a few variables that retain their values right through the run of the program.

PROCcreate is entered when you select the 'create' option at the start of the program. The table is cleared to 0's and various pointers are reset to indicate the first bar. PROCappend is then called to allow you to enter the bars.

PROCsetup. This one-liner sets up default values for replaying the music in case you do not set them up yourself.

PROCappend. This is the routine which allows you to enter the music. It checks for all the acceptable key-presses and calls PROCedures to deal with each one. You can only leave this routine by pressing X (exit) when you are at the start of an empty bar. You are not allowed to leave until the music has been tidied up. Similarly, the procedure to enter the note and play it will only do so if there is room for it in the bar.

PROCkeys draws the static part of the screen – i.e. the heading lines and the piano keyboard. This is only needed once while you are creating a tune as the bar itself is drawn by:

PROCstave. The bar lines are cleared and redrawn ready for a new bar to be entered.

PROCnewnote. Checks to see if the note you have just played can be fitted into the bar. If so it is entered by calling PROCupdate.

PROCdelete will delete the previous note in the bar and reset the various pointers.

PROCbarfull. When you press <RETURN> to enter a full bar, this PROCedure stores the displayed envelope and octave values with that bar and updates pointers to indicate the next bar. The new stave is drawn by calling PROCstave.

The next three PROCedures are all responsible for updating parts of the heading line. PROCnotelength, PROCenv and PROCoct will all print to the screen when called, but will only update the field in question if the parameter is non-zero. To print the field without updating it, the PROCedure is called with the parameter set to zero. The PROCedures are called in response to the L, V and S keys respectively.

PROCupdate enters the new note on to the screen, into your ears (it plays it) and into the table. The pointer to the current note is then updated.

PROCplay. This routine is called when you want to replay the tune. It is this piece of code that you will need to transfer to your own program so that it can play the tune. In fact you do not need all of the code, and about half of it can be omitted – more on this later.

PROCpn plays a note.

PROCedit. As the method of editing is different from that of entering notes it requires its own PROCedure, and this is it. When you select the number of the bar you want to edit, it is displayed with a pointer to indicate the note you are editing. To amend the details in the header line, the normal PROCedures are called, but when you enter a note, it will be inserted at the cursor position and overwrite the note (if any) that was there. The editing has been designed to allow you to enter anything, anywhere in the bar. However, you cannot enter the bar (by pressing <RETURN>) until it is valid; that is, the note lengths must add up to the time signature. When you move to a new note by pressing the < and > keys, PROCcursor is called to move the arrow.

PROCcursor overwrites the previous arrow and redraws it in its new position, while. . .

PROCarrow is called to do the actual printing.

PROCinsert replaces the note at the cursor by its new value and plays it. This routine does not update the cursor position.

PROCdeletenote. In the same way that the previous routine added a note without updating pointers, this PROCedure will remove the note at the cursor position.

PROCsave requires you to enter a title for your piece of music and then SAVEs it to the current filing system. Similarly:

PROCload asks you for the title of a tune which it will then locate and LOAD.

PROCtitle is used by both of the previous routines to get a title between one and seven characters long.

PROCvet is an important routine for checking all single key inputs. The parameter contains a list of valid replies and the routine exits when one has been selected. Variable F% contains the position of the reply in the parameter string.

PROCdraw enters the notes on to the stave. As the data is EOR'ed on to the screen, it can be erased by CALLing this PROCedure again without updating any pointers. The routine is messy because it has to deal with a number of different situations, such as rests, sharps and lines above or below the normal stave. (Why does music have to be so complicated?)

Finally, two functions are defined; FNfull returns the value TRUE or FALSE, depending on whether or not the current bar is full. FNnote reads a note value from the table. The note number within the bar is used as the parameter for the function, while the bar is taken to be the current bar. FNfull also serves one other purpose and that is to count the total length of all the notes and rests in the bar. This total is held in variable tot.

Variables

The list below shows the purpose of each of the important variables used by the program.

line$ – To determine whether the note should be drawn outside the normal stave and with its own line.

shrp$ – To determine whether the note you enter should be drawn with a '#' sign before it.

tpos$ – To convert each note to its true screen position (vertically).

Each of the above contains one entry for every note on the keyboard, which is accessed via the MID$ command.

keys$ – This string is used to validate all key presses from the main screen.

note$ – Contains the data for drawing the notes on the screen. Each note consists of two user-defined characters and it is printed by placing the first, following it with cursor down, cursor left and then printing the second. Each of the four notes comprises four consecutive bytes in this string.

len(6) – Holds the note lengths of all seven possible note types.

r1$ and r2$ are used to draw the piano keyboard. Each string contains the data for one of the seven (but only two different) screen lines that the keyboard occupies.

sig – The time signature of the current tune. It equals either 3 or 4.

nobars% – The number of bars in the piece (to date).

s% – The length of the note that will be entered when you are creating or editing a tune. This can be altered by pressing L.

env% – The ENVELOPE used to play the current bar.

oct% – The number of octaves by which the notes of the bar will be shifted (up) when the bar is played.

tempo – An indication of the speed of the music when it is played.

bar% – The number of the current bar.

ptr% – The screen position of the next note to be entered. It takes values from 1 to 8 inclusive.

L% and P% – Always appear together and are respectively the length and pitch of the current note (the one being played or entered). These values are not absolute as they are extracted from a byte in the table. Before the note is played they have to be further converted.

D% – the duration of the current note, ignoring the fact that it might be dotted. It is used to keep the screen in order.

temp – After the bar has been edited, it can end up in a real mess and so it is copied to a temporary storage area, then tidied up and copied back to the table. Next time you see the bar it will be neat and tidy again. This storage area is the eight bytes following temp.

A few other variables appear but their use should be reasonably obvious as most of them appear in just one PROCedure so that they are effectively LOCAL.

Extensions

This routine is a bottomless pit into which you can throw any number of ideas – the only limitation being the amount of memory available, and music is a sufficiently rich subject to ensure that you will eventually reach that limit. The utility presented here was written to help you create tunes for future use, rather than as an end in itself. It is very easy to get carried away and include all sorts of features that any eventually get too subtle for the routine that has to play the music!

Here are just a few ideas you might like to incorporate into the program, although this list is by no means definitive:

More envelopes, more time signatures, more note types, more voices, a bass clef, codas, rallentando, tied notes, portamento, auto-harmonies, a compose facility, key changes, phrasing, trills, dynamics, expression, and so on.

That little lot should keep you busy for a while! Actually, the last item should serve to remind you that computers are not really suitable for reproducing serious music and that we will never achieve perfect results however much we add to the routine.

One highly desirable feature that could easily be included is the addition of more voices. As written, the routine only allows you to create one line of music and you might like to amend it so that you can type in and edit two, or even three parts.

When you are working from written music, it is a straightforward task to enter and edit the parts separately and bring them together in a routine of your own that will play them. If you study the listing for the 'Duet' routine you will see that it takes two files (these are created by the main utility) and then plays the tune in two parts. This could easily be extended to incorporate a third part.

As you will see, the routine has been pirated from the main program and modified only slightly – notice though, how the data is loaded in lines 70–100.

Obviously, if your program only plays a single line of music, then the routine can be simplified even more. If the tune is only played once, then you do not need to have PROCplay as a procedure, and the variable tempo can be replaced by a constant.

The overall effect of the 'Duet' program is rather impressive, but if you intend to use the utility to create more complex music, you will have to amend it to allow the two parts to be played and edited together. This, in turn, might mean that some other compromise has to be made as memory is in short supply.

```
   10 REM DUET. The MUSIC HEADER program
 must have been used to set up envelopes
 before this will work.
   20 MODE 7
   30 HIMEM=&7000
   40 DIM len(6)
   50 FOR I%=0 TO 6:READ len(I%):NEXT
   60 DATA .5,.75,1,1.5,2,3,4
   70 tab1=&7001:tab2=&7401
   75 REM the data for these files is
given in the tables after this listing,
or you can create your own two part tune
   80 *LOAD "BACH-HI" 7000
   90 *LOAD "BACH-LO" 7400
  100 sig=tab1?-1:nobars%=?tab1
  110
  120 PROCplay(13)
  130 END
  140
  150 DEFPROCplay(tempo)
  160 tempo=22-tempo
  170 FOR bar%=1 TO nobars%
  180 oct1%=FNlead(9):env1%=FNlead(10)
  190 oct2%=FNbass(9):env2%=FNbass(10)
  200 FOR I%=1 TO 2*sig
  210 T%=TIME
  220 L%=FNlead(I%) DIV 32
  230 P%=FNlead(I%) MOD 32
  240 IF P% PROCpn(1,oct1%,env1%)
  250 L%=FNbass(I%) DIV 32
  260 P%=FNbass(I%) MOD 32
  270 IF P% PROCpn(2,oct2%,env2%)
  280 REPEAT UNTIL TIME>T%+2*tempo
  290 NEXT I%,bar%
  300 ENDPROC
  310
  320 DEFPROCpn(chan%,oct%,env%)
  330 IF P%=25 V%=0 ELSE V%=env%
  340 SOUND 16+chan%,V%,24*oct%+4*P%,len
(L%)*tempo
  350 ENDPROC
  360
  370 DEF FNlead(Z%)=tab1?(10*bar%-10+Z%
)
  380 DEF FNbass(Z%)=tab2?(10*bar%-10+Z%
)
```

```
0000 03 21 92 00 00 00 51 00 .!....Q.
0008 00 00 00 03 B2 00 00 00 ........
0010 00 00 00 00 00 03 AF 00 ........
0018 00 00 00 00 00 00 00 03 ........
0020 AD 00 00 00 00 00 00 00 ........
0028 00 03 B1 00 00 00 00 00 ........
0030 00 00 00 03 B2 00 00 00 ........
0038 00 00 00 00 00 03 51 00 ......Q.
0040 4D 00 4A 00 00 00 00 03 M.J.....
0048 4D 00 01 0B 0A 08 00 00 M.......
0050 00 03 8A 00 00 00 48 00 ......H.
0058 00 00 00 03 46 00 4A 00 ....F.J.
0060 46 00 00 00 00 03 AB 00 F.......
0068 00 00 00 00 00 00 00 03 ........
0070 4A 00 0B 0A 08 06 00 00 J.......
0078 00 03 88 00 00 00 45 00 ......E.
0080 00 00 00 03 86 00 00 00 ........
0088 4A 00 00 00 00 03 4B 00 J.....K.
0090 4D 00 4B 00 00 00 00 03 M.K.....
0098 AA 00 00 00 00 00 00 00 ........
00A0 00 03 A6 00 00 00 00 00 ........
00A8 00 00 00 01 A5 00 00 00 ........
00B0 00 00 00 00 00 01 43 00 ......C.
00B8 46 00 43 00 00 00 00 01 F.C.....
00C0 88 00 00 00 54 00 00 00 ....T...
00C8 00 01 A8 00 00 00 00 00 ........
00D0 00 00 00 01 4A 00 4D 00 ....J.M.
00D8 4C 00 00 00 00 01 4D 00 L.....M.
00E0 45 00 48 00 00 00 00 01 E.H.....
00E8 4D 00 41 00 59 00 00 00 M.A.Y...
00F0 00 01 59 00 8D 00 00 00 ..Y.....
00F8 00 00 00 04 59 00 8F 00 ....Y...
0100 00 00 00 00 00 04 52 00 ......R.
0108 51 00 52 00 00 00 00 04 Q.R.....
0110 11 0F 0D 0F 51 00 00 00 ....Q...
0118 00 04 59 00 59 00 45 00 ..Y.Y.E.
0120 00 00 00 04 46 00 4D 00 ....F.M.
0128 4B 00 00 00 00 04 4D 00 K.....M.
0130 4D 00 4B 00 00 00 00 04 M.K.....
0138 AA 00 00 00 00 00 00 00 ........
0140 00 04 00 00 00 00 00 00 ........
0148 00 00 00 FF ** ** ** ** ........
```

```
0000 03 21 4D 00 06 08 0A 0B .!M.....
0008 00 00 04 01 4D 00 46 00 ....M.F.
0010 46 00 00 00 04 01 4F 00 F.....O.
0018 0B 0D 0F 11 00 00 04 01 ........
0020 52 00 46 00 46 00 00 00 R.F.F...
0028 04 01 4B 00 0D 0B 0A 08 ..K.....
0030 00 00 04 01 4A 00 0B 0A ....J...
0038 08 06 00 00 04 01 45 00 ......E.
0040 06 08 0A 06 00 00 04 01 ........
0048 A8 00 00 00 00 00 00 00 ........
0050 04 01 4D 00 06 08 0A 0B ..M.....
0058 00 00 04 01 4D 00 46 00 ....M.F.
0060 46 00 00 00 04 01 4F 00 F.....O.
0068 0B 0D 0F 11 00 00 04 01 ........
0070 52 00 46 00 46 00 00 00 R.F.F...
0078 04 01 4B 00 0D 0B 0A 08 ..K.....
0080 00 00 04 01 4A 00 0B 0A ....J...
0088 08 06 00 00 04 01 48 00 ......H.
0090 0A 08 06 05 00 00 04 01 ........
0098 A6 00 00 00 00 00 00 00 ........
00A0 04 01 56 00 12 14 16 12 ..V.....
00A8 00 00 04 02 54 00 0D 0F ....T...
00B0 11 0D 00 00 04 02 52 00 ......R.
00B8 0F 11 12 0D 00 00 04 02 ........
00C0 4C 00 0A 0C 48 00 00 00 L...H...
00C8 04 02 08 0A 0C 0D 0F 11 ........
00D0 00 00 04 02 52 00 51 00 ....R.Q.
00D8 4F 00 00 00 04 02 51 00 O.....Q.
00E0 48 00 4C 00 00 00 04 02 H.L.....
00E8 AD 00 00 00 00 00 00 00 ........
00F0 04 02 4D 00 06 05 46 00 ..M...F.
00F8 00 00 04 04 4F 00 06 05 ....O...
0100 46 00 00 00 04 04 4D 00 F.....M.
0108 4B 00 4A 00 00 00 04 04 K.J.....
0110 08 06 05 06 48 00 00 00 ....H...
0118 04 04 01 03 05 06 08 0A ........
0120 00 00 04 02 4B 00 4A 00 ....K.J.
0128 48 00 00 00 04 02 0A 0D H.......
0130 46 00 45 00 00 00 04 02 F.E.....
0138 A6 00 00 00 00 00 00 00 ........
0140 04 02 00 00 00 00 00 00 ........
0148 00 00 00 FF ** ** ** ** ........
```

Utility 19:

Character definer

Description

Oh no! Not another VDU 23 character definer!

It had been my original intention to omit this particular routine, but as someone pointed out, you can hardly write a book of BBC utility programs and not include a version of this most popular of all utilities. Actually, the fact that this program has been written so many times illustrates that there is a need for utility programs on advanced computers where you can only get the best out of the machine by letting it help you do so.

As everyone is aware, the VDU 23 command enables you to create your own user-defined characters within an 8×8 matrix of dots (it has many other uses as well but let's not digress). Each line of the character is stored as one byte and the eight bits of that byte determine the configuration of the dots on that line. If you study the explanation of *FX20 in the *User Guide* you will see that any character in the range 32–255 can be redefined, provided that memory is reserved for the new definitions of characters in the range 32–223*. This is the approach we use and this utility enables you to redefine any character you like. Much fun can be had in swapping the alphabet for new characters!

After you have chosen an ASCII character number, the current definition of that character is reproduced on a large grid and it can then be edited in a number of ways. Also displayed are the images of that character in other MODES so that you have a good idea of what it will look like when displayed. At any time you may chose the option to display the entire character set so that you can see how your creative efforts are progressing. The various facilities offered by the program are listed in the next section.

Unlike some versions of this utility, it is only possible to create one character at a time, as most of the screen area is fully occupied.

Use

If you only want to experiment with the 'official' free characters (that is, those in the range 224–255), then you should delete line 50 and RUN the program in the normal way. However, to use the program as

---

*Note that early editions of the *User Guide* had a serious omission in the documentation of *FX20 and it is, in fact, necessary to use *FX20,6 to explode fully the character locations.

written, you must first set PAGE=PAGE+&600 before the program is LOADed. The free space between the usual value of PAGE and your program is used to hold the new definitions of characters with ASCII codes between 32 and 223.

Once the program is running, enter the number (32–255) of the character you want to edit. The current definition for that character is displayed (in three formats) and the following keys now come into effect:

Cursor keys: These keys move the cursor around the grid. The movement includes auto-repeat and full wrap-around.

Space bar: This is used to alter the status of the 'dot' at the cursor position, so that a dot becomes a space, and vice versa. The cursor is moved one place to the right so that the next dot can be amended.

D: Displays the full character set. Each line contains 32 characters, which correspond to a full Page in the reserved area.

E: Empty the grid. The entire grid is wiped ready for you to edit the character.

F: Fill the grid. If you are going to define a character that contains more dots than blanks, it makes sense to start off with the display full of dots so that you only need to insert the blanks.

I: Invert the status of each bit within the grid so that dots become spaces and spaces become dots. If you do this with the letters of the alphabet you will achieve the 'reverse field' effect which is included as standard on some computers.

N: Proceed to edit a new character. If you take this option, the character that you have just edited is not saved and will be lost. If you mean to save a character, you must specifically say so.

Q: Quit the program.

S: Save the character currently on display. The VDU command required to set up that character is displayed on the bottom line of the screen and then executed. Following this, the cursor is returned to the grid to allow further editing.

While the character is being amended, the MODE 1 and MODE 2 versions are also updated so that all three images show the same character.

```
   10 REM (ANOTHER) VDU 23 CHARACTER DEF
INER
   20 MODE 1
   30 DIM A%(7)
   40 *FX 4,1
   50 *FX 20,6
   60 REMOVE ABOVE LINE UNLESS PAGE AUGM
ENTED BY &600
   70 REPEAT
   80 REPEAT
   90 CLS
  100 INPUT'"ASCII code "char%
  110 UNTIL char%>31 AND char%<256
  120 PROCscreen
  130 PROCreaddots
  140 PROCfill(255,0)
  150 REPEAT
  160 F%=INKEY(0)
  170 IF F%=32  PROCflip
  180 IF F%=68  PROCdisplay
  190 IF F%=69  PROCfill(0,0)
  200 IF F%=70  PROCfill(0,255)
  210 IF F%=73  PROCfill(255,255)
  220 IF F%=83  PROCsave
  230 IF F%=136 PROCmove(-1,0)
  240 IF F%=137 PROCmove(+1,0)
  250 IF F%=138 PROCmove(0,+1)
  260 IF F%=139 PROCmove(0,-1)
  270 UNTIL F%=68 OR F%=78 OR F%=81
  280 UNTIL F%=81
  290 CLS
  300 *FX 4,0
  310 END
  320
  330 DEFPROCscreen
  340 GCOL 0,2
  350 PRINT TAB(0,6)"MODES 1,4,6:
 MODES 2,5:"''
  360 PRINT'"D - Display all"
  370 PRINT'"E - Empty"
  380 PRINT'"F - Fill"
  390 PRINT'"I - Invert"
  400 PRINT'"N - New char"
  410 PRINT'"Q - Quit"
  420 PRINT'"S - Save char"
  430 FOR I%=592 TO 1104 STEP 64
  440 MOVE I%,208:DRAW I%,720
  450 NEXT
  460 FOR I%=208 TO 720 STEP 64
  470 MOVE 592,I%:DRAW 1104,I%
  480 NEXT
  490 ENDPROC
  500
  510 DEFPROCreaddots
  520 LOCAL A%,X%,Y%
  530 A%=10:X%=&80:Y%=0
  540 ?&80=char%
  550 CALL &FFF1
  560 FOR I%=0 TO 7
  570 A%(I%)=I%?&81
  580 NEXT
  590 ENDPROC
  600
  610 DEFPROCfill(U%,V%)
  620 FOR Y%=0 TO 7
  630 A%(Y%)=(A%(Y%) AND U%) EOR V%
  640 FOR X%=0 TO 7
  650 PROCpixels
  660 NEXT,
  670 X%=0:Y%=0
  680 PROCmove(0,0)
  690 ENDPROC
  700
  710 DEFPROCpixels
  720 K%=828-4*Y%
  730 IF A%(Y%) AND 2^(7-X%) GCOL 0,1 EL
SE GCOL 0,0
  740 MOVE 596+64*X%,662-64*Y%
  750 MOVE 596+64*X%,716-64*Y%
  760 PLOT 81,56,-56
  770 PLOT 81,0,56
  780 PLOT 69,464+4*X%,K%
  790 PLOT 69,470+4*X%,K%
  800 PLOT 69,1040+8*X%,K%
  810 PLOT 69,1044+8*X%,K%
  820 PLOT 69,1048+8*X%,K%
  830 ENDPROC
  840
  850 DEFPROCmove(U%,V%)
  860 X%=(X%+U%+8) MOD 8
  870 Y%=(Y%+V%+8) MOD 8
  880 VDU 31,2*X%+19,2*Y%+10
```

```
 890 ENDPROC
 900
 910 DEFPROCflip
 920 A%(Y%)=A%(Y%) EOR 2^(7-X%)
 930 PROCpixels
 940 PROCmove(+1,0)
 950 ENDPROC
 960
 970 DEFPROCdisplay
 980 CLS
 990 PRINT TAB(13)"Character set"'''
1000 FOR I%=32 TO 224 STEP 32
1010 PRINT''TAB(4);
1020 FOR J%=0 TO 31
1030 IF I%+J%=127 VDU 32 ELSE VDU I%+J%
1040 NEXT,
1050 PROCrastinate
1060 ENDPROC
1070
1080 DEFPROCsave
1090 PROCvducodes
1100 VDU 23,char%,A%(0),A%(1),A%(2),A%(
3),A%(4),A%(5),A%(6),A%(7)
1110 PROCmove(0,0)
1120 VDU 7
1130 ENDPROC
1140
1150 DEFPROCvducodes
1160 VDU 28,0,29,39,28,12
1170 PRINT "V.23,";char%;
1180 FOR I%=0 TO 7
1190 PRINT",";A%(I%);
1200 NEXT
1210 VDU 26
1220 ENDPROC
1230
1240 DEFPROCrastinate
1250 *FX 15,1
1260 PRINT TAB(8,31)"Press SPACE to con
tinue";
1270 REPEAT UNTIL GET=32
1280 ENDPROC
```

How it works

Once you have chosen the character number, its current definition is read into an array using an OSWORD call with A=10. The character is displayed in its various formats and the program enters a loop from which a PROCedure is called to perform the selected option. To exit the loop you must type Q, which terminates the program.

Procedures

Once you have chosen a character number, PROCdisplay is called to create the initial screen – this PROCedure will not be called again until a new character is displayed. Similarly, PROCreaddots is called only once to fetch the current status of the selected character.

PROCfill is used to reset the character definition, whatever the nature of the redefinition is. To do this, it takes two parameters that are ANDed and EORed with the eight bytes that define the character to compute the new bytes, and hence the new display. This may seem like a strange way of setting the bits but it is actually very sensible; the thinking behind it will be found – in a different context – in a section in the *User Guide* describing OSBYTE calls &E1 to &E8. Once the bytes that define the character have been amended, this PROCedure calls PROCpixels to reproduce the effect on the screen.

Whenever a cursor key is pressed, PROCmove is called with two parameters (across and down) to shift the cursor. This routine may also be called by others to restore the cursor to the grid area.

The spacebar controls the status of each dot and to do so it calls PROCflip so named because it flips the status of the bit (i.e. dot) at the cursor. If you find it annoying that the cursor is automatically stepped to the right, delete the call to PROCmove at line 940. PROCdisplay and PROCsave are used to perform their eponymous actions in response to the D and S keys respectively. When PROCsave is called, its first action is to call PROCvducodes which gives the VDU 23 command to create that character. VDU has been abbreviated to V, which is acceptable to BASIC, so that the command will fit on to one line.

Finally, PROCrastinate (geddit?) is the delay routine which only returns after you have pressed the spacebar. Nice piece of coding – shame about the joke!

Variables

The most important variable used by the routine is the array A%(7); it is into this array that the eight bytes defining the character are read. Subsequent editing will amend the contents of the array so that it is consistent with the character displayed on the screen at all times. The ASCII code of the character being amended is referred to throughout as char%.

In common with many programs that move items around the screen, X% and Y% are used to indicate the screen co-ordinates of the cursor. In fact the values are not absolute but relative to the top left-hand corner of the grid area: these two variables effectively define the position within the grid rather than on the screen.

Extensions

Since this is the most written utility of all, it is hardly surprising that there are numerous variations on it. Some of these allow rotation and reflection of the displayed shape (although personally, I think even 'Invert' is going a bit far) and many allow multiple characters to be defined, say up to a 4*3 block. The latter addition is a useful one if you intend to build very large shapes from several characters – you can then use the screen almost as a 'sketch-pad' to create them.

One very useful addition that is easily included (have a go at this yourself) is to allow the contents of the character storage areas to be SAVEd. Because the areas are separated, they should be SAVEd in two parts: one page from &C00 (for characters 224–255) and six pages from the original value of PAGE (for the rest). These commands will do the job:

```
*SAVE C00+100
*SAVE E00+600
```

Disc owners should use 1900 instead of E00 in the last command. The program should also include a PROCedure that can *LOAD the areas back for future use.

If you like the idea of 'exploding' the character set but do not want the luxury of of being able to redefine every character, then you may wish to use a lower *FX20 command at line 50 – say *FX20,1. If you look at the write-up on this command, you will see that this only requires one page to be reserved instead of the six we have used.

Utility 20:

Screen save

Description

Having created a complex screenful of graphics, you may wish to SAVE the contents of the screen to tape or disc and this short routine enables you to do just that. This is particularly convenient with discs because a picture can often be LOADed from disc much quicker than it can be drawn in BASIC. Although a cassette takes longer to LOAD, the value of being able to SAVE a screen – either complete, or so that it may be further developed – is immense.

Two versions of the utility are presented: a function key version that works in immediate mode, and a PROCedure that can be called from within a BASIC program. The utility is in two parts: the SAVE routine itself, and a corresponding LOAD routine.

In MODE 0, 1 or 2, the routine has to save 20000-odd bytes and this takes some time – of the order of three seconds for disc systems, and four and a half minutes for cassettes.

Use

Having established the screen following a MODE change or a CLS, it is important to ensure that it does not scroll before it is SAVEd. To SAVE the screen from within a program, include PROCsave with that program and call it when required. If you are working in 'immediate' mode, simply press function key f0. In either case, operate the cassette recorder to correspond with the 'cassette motor' light – you will not get any tape messages. Remember to start recording when the motor light appears, then press <RETURN>. Disc users will find that the discs start and stops automatically.

When the screen is SAVEd using PROCsave or key f0, it can be reloaded by either PROCload or function key f1, as appropriate.

```
   10 REM SAVE/LOAD SCREEN BY KEYS OR PR
OCS
   20
   30 *KEY0 ¦O¦\¦@¦@¦@¦@$&900="S. S "+ST
R$~H.+" 7FFF":X%=0:Y%=9:CA.&FFF7¦M¦Z¦G
   40 *KEY1 ¦O¦\¦@¦@¦@¦@*L."S"¦M¦Z¦G
   50
   60 END
   70
```

```
 80 DEFPROCsave
 90 VDU 15,28,0,0,0,0
100 $&900="SAVE S "+STR$~HIMEM+" 7FFF"
110 X%=0:Y%=9
120 CALL &FFF7
130 VDU 26,7
140 ENDPROC
150
160 DEFPROCload
170 VDU 15,28,0,0,0,0
180 *LOAD S
190 VDU 26,7
200 ENDPROC
```

How it works

The screen area is treated as a section of memory and is simply *SAVEd. In the Block Move utility, we saw that the memory allocated to the screen starts at HIMEM but HIMEM does not necessarily correspond to the top left-hand corner of the screen display. This is because of the hardware scrolling used by the computer. The (whole) screen will always be SAVEd, but on reLOADing, it will appear to be displaced if a scroll has taken place.

To avoid this problem it is important to SAVE the screen before it gets the chance to scroll. For most applications this is a sensible requirement, anyway.

When f0 is pressed, page mode is cancelled and a zero-sized text window is established so that any subsequent printout does not disrupt the screen. After the screen RAM has been *SAVEd, the computer bleeps and default windows are restored. You can retrieve the cursor by pressing <RETURN>.

The use of a text window ensures that no messages can be printed, but it also means that the first character of the stored screen will be a space. This is slightly annoying but it is necessary to print something, somewhere, before the window can be created. With the PROC version, this problem can be avoided by amending lines 90 and 170 :for discs, simply delete them and for tapes replace them by *OPT 1, 0 (abandon cassette messages). If this is done, the SAVEd screen is exactly as seen, with the first character intact.

By far the best place for the text window is down at the bottom right hand corner of the screen, where it is difficult to PRINT at the best of times. Unfortunately. this position depends on the screen MODE and is not constant. This in turn means that the routine would need to do MODE checks and the coding would be much more complicated. It hardly seems worth it to avoid the small inconvenience of having the first character blanked out.

Notice that the *SAVE line contains variable information, namely HIMEM, and so it is necessary to assemble the string into store – we use &900 – and call the 'Command Line Interpreter' to execute it. The resulting file is given the title 's'. If you have BASIC 1982 you can use the command OSCLI(a$) to pass a command to the 'Command Line Interpreter'.

Procedures

The two procedures have been coded to perform exactly the same functions as the key versions. This should make the rather mysterious key coding easier to understand. To save space in the function key area, the '¦' operator has been used extensively to set up control codes and this does make the definitions difficult to follow.

Extensions

The scrolling problem could be dealt with in several ways:

i) Don't scroll!
ii) Read the screen using the OSWORD 'read pixel' call and save it as a file.
iii) For the adventurous only: read the address of the first byte of the screen from locations &350/1 and make allowances for it, either before or after the screen is SAVEd. Best of luck with this one!

If you save the screen as a file (a series of bytes BPUT to tape or disc), then it is possible to SAVE the current MODE along with it and then the screen can be forced to reLOAD correctly. Again, this is rather a luxury and anyway it is possible to obtain some different (one refrains from saying 'pleasant') effects by reLOADing in the 'wrong' MODE.

One of the great advantages of this routine is its simplicity and any amendments should retain this virtue as far as is practical.

This routine is obviously not meant to be used on its own, but would make a useful addition to any program that created graphics displays – for example, some of those in this book!

Utility 21:

MODE 2 character creator

Description

Video games are the most popular type of programs for home computers and yet very little information is available about the way they are written. This is partly because, for speed reasons, such games have to be written in machine code, but mostly because it is more profitable to write the games than it is to write about how to program them!

Here we present a couple of utilities that are a must if you intend to write any video games for the BBC computer. The second utility opens up the possibility of writing fast moving arcade-type games in BASIC and should appeal to all readers. Both utilities (they are complementary and should be used together) assume that your program will use MODE 2 so that the full range of colours is available. If you wish to use other MODES (which probably means 5 for chess and 1 for everything else) you should enjoy doing the appropriate research to enable you to modify the programs.

To generate high resolution multicoloured shapes on the screen, it is fastest and easiest to address the screen directly – a technique with the dubious title of 'POKEing' when applied to other computers. Officially you are not supposed to do this because your programs will not work in conjunction with the second processor that may be fitted to the machine. No doubt this option is a wonderful thing, but it's a safe bet that you haven't got one and you should certainly not lose any sleep over POKEing a few bytes on to the screen here and there.

Obviously, before we can address the MODE 2 screen, it is necessary to know its layout (memory-map), and this is described below.

The MODE 2 screen is divided horizontally into 32 lines and each line is further divided into 80 'characters' consisting of 8 pairs of adjacent dots arranged to form two columns. Note that these are not the normal characters that you get in MODE 2 – there are only 20 of those to a line.

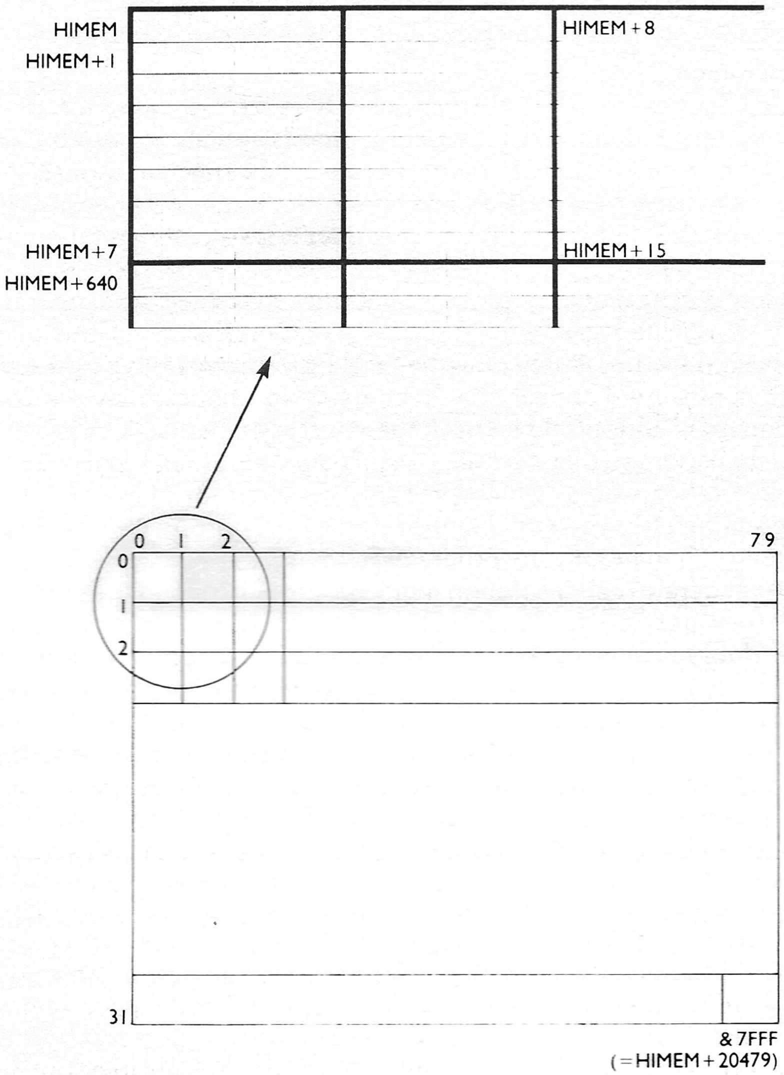

HIMEM
HIMEM+1
HIMEM+7
HIMEM+640
HIMEM+8
HIMEM+15
0
1
2
79
0
1
2
31
&7FFF
(=HIMEM+20479)

Fig. 4. Memory map of MODE 2 screen.

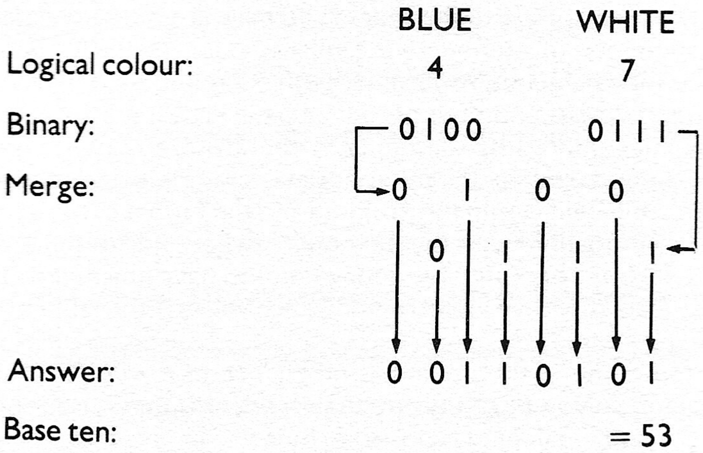
POKEing, or storing data into HIMEM, HIMEM+1 etc. will cause one or both dots (depending on the data) to light up in the first character position. Locations HIMEM+8 through HIMEM+15 define the second character and so on across the first row. After 80*8 bytes, we reach the second row and the first pair of dots on that row are at HIMEM+640.

Thus, to illuminate the character at column X (0<=X<=79) and row Y (0<=Y<=31), we must POKE location:

HIMEM + 8*X + 640*Y

and the seven bytes following it.

The data required to create a particular effect is related to the pattern of dots in a rather strange way, best illustrated by means of an example. Suppose we wish to light up two adjacent dots as blue/white, the calculation would be:



Quite a tedious process!

This utility performs all of these calculations automatically and includes edit and SAVE facilities as well. The product is the data, either on file or displayed on the screen, to create MODE 2 characters between two and twenty-four dots wide and one row deep. Thus, it is ideal for creating all sorts of nasty little aliens for your space games, but it also has more friendly, down-to-earth uses.

This program enables you to set up the data in a sensible way but does not get the character on to the screen for you (except for preview purposes) – the next utility picks up from here and gets them shifting around the screen in double-quick time!

Use

The program is written in BASIC and can simply be RUN.

Initially, you will have to create a character, so select the 'Create' option. Choose the width of your 'alien', which must be an even number of dots between two and twenty-four inclusive – as a guide, the width of a text character in MODE 2 is eight dots.

Using the cursor keys you can move around the grid in the usual way (wrap-around, auto-repeat) and to colour in a dot, press the key corresponding to the colour, as indicated by the menu at the bottom of the screen. Notice that black never gets PRINTed as '0', but is always left blank as it is easier to see the outline of the shape if this is done. Use CTRL X to get out of this mode back to the main menu.

You will probably want to see the creature (no doubt that's what it will be!) in glorious technicolour, so choose the 'View shape' option and you will see how the character looks in MODE 2. If the beast is not to your liking, entering 'Edit' mode will enable you to alter it using the same facilities as 'Create'.

Once you are happy with the shape, you may 'List numeric data' to see the sequence of numbers you must POKE on to the screen to reproduce the character. You can write these numbers down but if you intend to devise a number of characters, it is best to use another of the facilities offered by the utility, namely 'Store numeric data'. When this option is taken, the numeric data – along with its length – is stored in a table held within the program and the format of the data in the table is compatible with the next utility, where it is described in more detail. Look after this table and, when you have finished adding data to it, select the 'Save table' option which will SAVE the table to the current filing system.

As presented here, the utility has no chance of reading the table back in again, so you must be sure that the data within it is more or less correct (some patching is possible) before you save it.

```
   10 REM MODE 2 ALIEN CREATOR
   20 MODE 3
   30 *FX 4,1
   40 DIM A%(12,8),array%(15),table% 100
0
   50 FOR I%=0 TO 15
   60 READ array%(I%)
   70 NEXT
   80 ptr%=0
   90 REPEAT
  100 PROCmenu
  110 IF M%=1 PROCcreate:PROCedit
  120 IF M%=2 PROCedit
  130 IF M%=3 MODE 2:PROCview:MODE 3
```

```
140 IF M%=4 PROClist
150 IF M%=5 PROCstore
160 IF M%=6 PROCsave
170 UNTIL M%=7
180 *FX 4,0
190 END
200
210 DEFPROCmenu
220 VDU 26,12,7
230 PRINT''" Options:"'
240 PRINT TAB(7)"1) Set up new pattern
"
250 PRINT TAB(7)"2) Edit existing patt
ern"
260 PRINT TAB(7)"3) View shape"
270 PRINT TAB(7)"4) List numeric data"
280 PRINT TAB(7)"5) Store numeric data
"
290 PRINT TAB(7)"6) Save table"
300 PRINT TAB(7)"7) Quit program"
310 REPEAT
320 INPUT'"Your choice "M%
330 UNTIL M%>0 AND M%<8
340 ENDPROC
350
360 DEFPROCcreate
370 REPEAT
380 INPUT'"Width of shape(even) "W%
390 UNTIL (W%>1) AND (W%<25) AND (W% A
ND 1)=0
400 FOR I%=1 TO W%/2:FOR J%=1 TO 8
410 A%(I%,J%)=0
420 NEXT,
430 ENDPROC
440
450 DEFPROCedit
460 PROCboard
470 X%=0:Y%=0
480 VDU 28,tab%+2,14,tab%+3*W%,7
490 REPEAT
500 VDU 31,3*X%,Y%
510 F%=INKEY(0)
520 IF F%=136 X%=(X%-1+W%) MOD W%
530 IF F%=137 X%=(X%+1) MOD W%
```

```
  540 IF F%=138 Y%=(Y%+1) MOD 8
  550 IF F%=139 Y%=(Y%+7) MOD 8
  560 IF (F%>47 AND F%<58) OR (F%>64 AND
 F%<71) PROCtrish
  570 UNTIL F%=24
  580 ENDPROC
  590
  600 DEFPROCtrish
  610 IF F%>64 byte%=F%-55 ELSE byte%=F%
-48
  620 IF F%=48 F%=46
  630 mask%=85:factor%=2
  640 IF (X% AND 1) mask%=170:factor%=1
  650 was%=A%((X% DIV 2)+1,Y%+1) AND mas
k%
  660 A%((X% DIV 2)+1,Y%+1)=was%+factor%
*array%(byte%)
  670 VDU F%
  680 X%=(X%+1) MOD W%
  690 ENDPROC
  700
  710 DEFPROCboard
  720 CLS
  730 PRINT TAB(19,4)"***  Press CTRL X
to recall menu ***"
  740 tab%=(72-3*W%) DIV 2
  750 PRINT TAB(0,6)
  760 FOR J%=1 TO 8:PRINT TAB(tab%);
  770 FOR I%=1 TO W%:V%=A%((I%+1) DIV 2,
J%)
  780 mask%=85:factor%=1
  790 IF (I% AND 1) mask%=170:factor%=2
  800 V%=(V% AND mask%) DIV factor%
  810 K%=-1
  820 REPEAT:K%=K%+1:UNTIL array%(K%)=V%
  830 IF K%=0 A$="." ELSE A$=CHR$(48+K%-
7*(K%>9))
  840 PRINT "  ";A$;
  850 NEXT I%:PRINT:NEXT J%
  860 PRINT TAB(0,20);
  870 RESTORE 1340
  880 FOR I%=1 TO 16
  890 READ X$:L=LEN X$
  900 PRINT X$;SPC(20-L);
 910 NEXT
 920 ENDPROC
 930
 940 DEFPROCview
 950 FOR I%=1 TO W%/2:FOR J%=1 TO 8
 960 ?(HIMEM+3431+8*I%+J%)=A%(I%,J%)
 970 NEXT,
 980 PROCrastinate
 990 ENDPROC
1000
1010 DEFPROClist
1020 PRINT''"THE SEQUENCE OF NUMBERS YO
U NEED IS:"''
1030 FOR I%=1 TO W%/2:FOR J%=1 TO 8
1040 PRINT STR$(A%(I%,J%));",";
1050 NEXT,
1060 VDU 127
1070 PROCrastinate
1080 ENDPROC
1090
1100 DEFPROCstore
1110 table%?ptr%=4*W%+1
1120 FOR I%=1 TO W%/2:FOR J%=1 TO 8
1130 ptr%=ptr%+1
1140 table%?ptr%=A%(I%,J%)
1150 NEXT,
1160 ptr%=ptr%+1
1170 ENDPROC
1180
1190 DEFPROCsave
1200 X=OPENOUT("scrdata")
1210 FOR I%=0 TO ptr%
1220 BPUT# X,table%?I%
1230 NEXT
1240 CLOSE# X
1250 ENDPROC
1260
1270 DEFPROCrastinate
1280 PRINT''"Press C to continue";
1290 REPEAT UNTIL GET$="C"
1300 ENDPROC
1310
1320 DATA 0,1,4,5,16,17,20,21,64,65,68,
69,80,81,84,85
```

```
 1330
 1340 DATA "0=BLACK",4=BLUE,8=BLACK/WHIT
E,C=BLUE/YELLOW
 1350 DATA "1=RED",5=MAGENTA,9=RED/CYAN,
D=MAGENTA/GREEN
 1360 DATA "2=GREEN",6=CYAN,A=GREEN/MAGE
NTA,E=CYAN/RED
 1370 DATA "3=YELLOW",7=WHITE,B=YELLOW/B
LUE,F=WHITE/BLACK
```

How it works

In common with a number of our utilities, the overall structure of this program is very simple. Following a short initialisation section, several separate PROCedures can be summoned from a short loop which controls the flow of the program. When the loop ends, so does the program. A more comprehensive description of how it all works will be found in the next two sections.

Procedures

When you first enter the program, or after you have used one of its facilities, PROCmenu is called to PRINT a list of the available options and to receive your selection. If you choose the 'Create' option, then PROCcreate is entered to accept the width of the shape and to initialise the array A%(12,8) (or at least as much of it as you will use) and the main PROCedure, PROCedit is called. This PROCedure controls movement around the screen and is responsible for updating the array A% so that it reflects the state of the screen at all times. This is a complicated process and PROCtrish is used to assist with the calculations.

PROCboard sets up the screen whenever a character is to be edited or created. Its most difficult task is to decipher the array A% into patterns of colour, which is exactly the reverse of the steps carried out in PROCtrish.

The remaining PROCedures are quite straightforward and each is called in response to a selection from the menu. Before PROCview can be called, it is necessary to engage MODE 2 so that the data bytes can be POKEd on to the screen. On leaving this PROCedure MODE 3, which is the natural MODE for this utility, is then restored. PROClist is used to display the contents of the array, while PROCstore is used to commit them to memory (the computer's – not yours). Finally, PROCsave stores the contents of the table, as single bytes, to the filing system, be it tape or disc.

Variables

The merging process described earlier is rather fiddly and involves converting numbers into binary and re-organising their bits. To speed up the process an array is used to hold the converted versions of each number (or colour) in the range 0–15. This array is called array%(15) (where does he get these snappy names from?) and is initialised at the very start of the program. Swapping n for array%(n) is equivalent to the step from 'Binary' to 'Merge' in the example given in the Description section.

The array A%(12,8)has already been mentioned – it is the computer's version of what you see on the screen. When you choose to store the character you have just created, it is saved in a table whose first byte is known as table%, and which extends for 1000 bytes; it is a protected area and will not be overwritten. To indicate the extent of the table, ptr% points to the last byte entered into it.

As usual, X% and Y% represent the position of the cursor in the grid displayed on the screen (rather than the actual screen location, which is a function of both X% and Y%). The overall width of your character, in dots, is represented by W% – since the dots occur in pairs this will always be an even number.

When a dot has its colour changed, the heavy stuff starts (lines 630–660) and what happens here is this: If you alter the left-hand dot of a pair, for example, the right-hand colour is extracted (line 650) and remembered as was%. Based on the new colour (indicated by byte% – the colour you type-in in the range 0 – 15), the array entry for those two dots is recalculated by inserting the new value for the amended dot (line 660). Two new variables are introduced to help with this, namely: mask% which is either 85 (binary 01010101) or 170 (binary 10101010) and factor% which is either 1 or 2. Both mask% and factor% are used again by the PROCedure that displays the pattern of dots on the screen. As used by this PROCedure, they decode the bytes of the array rather than encode them.

Assorted localised (but not LOCAL) integer variables are used for temporary storage and the only other named variable is tab% which is calculated by PROCboard to help to centralise the grid on the screen whatever character width is selected.

Extensions

One function lacking from this program is the ability to edit the table of data bytes. Because of this, I have not even included a section to read in the table from tape or disc. This is not a difficult task but it will certainly increase the length of the program (which, in turn, means that you cannot allocate more bytes to the table which is possibly a more useful amendment). The real problem arises when you want to delete a shape from the table, because it will then be necessary to reposition the data remaining in it.

Because of the organisation of the screen into 'lines' the tallest character can only be one line, or eight dots high, and larger shapes will have to be made up of two or more standard characters. As the shape is deposited onto the screen in machine code, time is not a problem and the coding required to handle tall characters is hardly more complicated than that for the standard size.

When you feel that you have mastered this program, why not write a similar routine (or cannibalise this one) to enable you to create characters in MODE 10 Before doing so you will need to investigate the layout of the screen in that MODE – there are plenty of clues in the description of this program.

Before proceeding to the next utility, it is a good idea to get some data ready for it. Choose a width of 10 (a good size for your common alien) and firstly create a blank character – that sould not cause too many problems. Now invent two or three other shapes, also of width 10. When you have done so (don't forget to store each pattern into the table), save the table onto tape.

Now that you have some characters on tape, let's get them onto the screen.

Utility 22:
MODE 2 character plotter

Description

The previous utility enables you to set up data which, when POKED onto a MODE 2 screen, will draw a character of your own design. This program – which is not a utility as such –will get that character onto the screen very quickly as it is written in machine code, although it can be called from both BASIC and machine code routines. If your aliens are lacking zip, this is just the thing you need to get them pinging round the screen at high speed.

This routine expects the data to be set up in a special (and, as we shall see, familiar) way, which by a happy coincidence is exactly how the previous utility does it. What luck!

There are two parts to this routine: the data has to be found and extracted from the table, then it has to be deposited at the correct screen location. Let us attend to these in turn.

The table you have SAVED on tape or disc contains the information to draw a few characters onto the screen; this program will think of them as character 1, character 2, and so on. To locate character 3, say, we start at the beginning of the table and pick up the byte there – suppose it is 41 (which it will be if you followed the advice at the end of the last utility; although it doesn't matter too much if you didn't). This number represents the number of bytes, including itself, allocated to the first character and by adding it to the address of the start of the table, you will produce a new address, namely that of the next character. Again, the first byte will represent the length of the data associated with that character, so we can add once more to get the address of the character we are looking for. All the time this stepping process is going on, we count down the number of the character we are searching for until it hits zero indicating that the search is complete.

Where have you seen this before? Here's a clue –look at the section on the storage of BASIC programs in Section 3. Basically (that's a dreadful pun and not at all intentional) we locate our aliens in the same way that BASIC finds GOTO or GOSUB line numbers. If you want to be ultra-efficient try to make sure that important, much-used characters occur at the beginning of the list so that the routine doesn't need to waste its time (and here we are talking about millionths of a second) looking for them. The data appears in the table like this:

table %

41	40 data bytes for Shape 1	25	24 data bytes for Shape 2	61	60 data bytes for Shape 3

Once the item has been located, the first data byte indicates how many bytes are to be printed to produce that shape; in the example above, 40 bytes are to be sent to the screen.

So, step one has been completed – first find your alien, as they say. Now we must see how to get it (him? her? – who knows with aliens?) onto the screen.

As far as our characters are concerned the effective resolution of the screen is 80 across by 32 down. For super-smooth vertical movement, the character should ideally be capable of spanning two lines, but this would need a great deal of extra coding, although the same data could still be used. In practice, this resolution has proved to be quite sufficient for all of the usual types of game.

To draw the shape we need to know the location of its first character that is, set of dot-pairs, which can be determined in X and Y co-ordinates, taking the origin to be at the top left-hand corner of the screen, at HIMEM. The start location for the character will then be HIMEM + 8*X + 640*Y – an expression we have already come across. The routine does this calculation and stores the result in Zero-page for future use. To print the character, we only need to poke bytes into this and the following locations until all are used up.

Use

The routine has been written to make it as easy to use as is reasonably possible. Once the code has been safely assembled the program itself can be deleted, if you so wish – obviously you will make sure that it works first. Before running the program, amend line 40 so that base points to the start of your table of data bytes. You should have this table on tape, so get it into store with a *LOAD command, choosing a safe place to load it. As HIMEM will be at &3000 for MODE 2, a suitable place would be &2C00, so use the command:

```
*LOAD "scrdata" 2C00
```

Before going any further it is a good idea to protect this with HIMEM=&2C00.

Change line 40 to read:

```
40 base=&2C00
```

and amend line 70 if you wish to store the code at some place other than &900. The code can be called from BASIC or machine code and needs three parameters, passed to it via the accumulator, X and Y registers as shown:

Machine Code	BASIC	**Purpose**	**Range**
Accumulator	A%	Character number	1 – number of items in table
X-register	X%	Start coloumn	0 – 79
Y-register	Y%	Start row	0 – 31

After the shape has been displayed, control is returned to you. If you are in machine-code, both the X and Y registers will contain 0 and the accumulator will be indeterminate.

Machine coders should have little difficulty in getting this routine up and running; for the BASIC programmer, here is some supplementary information.

What you do with the co-ordinates is your business, but once they have been calculated you can call the routine to draw the shape for you. In the last section of the previous utility I suggested saving a blank character because the easiest way to move things around is to blank the old shape out and rewrite it in a new position. This may seem like an extremely unsophisticated way of doing things but you must remember that the hard work is done by the machine code and it is exceptionally fast. In general, this system will always work, although for specific games, it may be better to create shapes that wipe themselves out as they move (for example surround them with two blank characters).

The short piece of code below will make alien number 2 fly across the 21st line of the screen. See if you can follow how it works for yourself.

```
10 MODE 2
20 X%=0:Y%=0
30 REPEAT
40 A%=1 : CALL &900
50 X%=X%+1
60 A%=2 : CALL &900
70 wait=INKEY(7)
80 UNTIL X%>74
```

Here I have assumed that you have stored a blank character for alien number 1. If you want your alien to scream across the screen instead of merely flying, try deleting line 70, or at least amending the INKEY parameter to 1 or 2. This is only a very simple example but it illustrates how fast multicoloured movement can be achieved from BASIC – you can concentrate on the rest of the program without having to worry about the screen handling.

This final, simple example shows you how to get your aliens into the traditional battle formation:

```
10 MODE 2
20 FOR Y%=4 TO 16 STEP 3
30 READ A%
40 FOR X%=0 TO 60 STEP 6
50 CALL &900
60 NEXT : NEXT
70 END
80 DATA 2,3,3,4,4
```

If you go to the trouble of making this work (your data tape will need 4 aliens on it, at least) I guarantee that you will be impressed by the speed with which the screen is set up – it is virtually instantaneous.

```
   10 REM MODE 2 ALIEN PLOTTER
   20 addrlo=&80:addrhi=&81
   30 scrnlo=&82:scrnhi=&83
   40 REM ***** SET base TO POINT TO TAB
LE *****
   50 FOR I%=0 TO 2 STEP 2
   60 REM SET P% TO SUIT - FOR EXAMPLE..
....
   70 P%=&900
   80 [OPTI%:
   90 .L0 PHA                ;remember shape
 number
  100      TYA               ;get row
  110      ASL A             ;x2 (NB.  C=0)
  120      TAY
  130      LDA &C376,Y       ;get low byte f
rom x640 table
  140      STA scrnlo        ;remember it
  150      LDA &C375,Y       ;fetch high byt
e
  160      ADC #&30          ;+HIMEM for MOD
E 2
  170      TAY               ;YR=high byte
  180      TXA               ;get column
  190      ASL A
  200      ASL A             ;x 4
  210      BCC L2
  220      INY               ;augment high b
yte...
  230      INY               ;...twice
  240 .L2 ASL A          ;x 2 more gives
 a total of x 8
  250     BCC L3
  260     INY            ;augment high
  270     CLC
  280 .L3 ADC scrnlo     ;add in low byt
e
  290     BCC L4
  300     INY
  310 .L4 STA scrnlo     ;store low
  320     STY scrnhi     ;and high
  330     PLA            ;recall shape n
umber
  340     TAX            ;use XR to coun
t down
  350     LDA #(base MOD 256)   ;point t
o start of table
  360     STA addrlo
  370     LDA #(base DIV 256)
  380     STA addrhi
  390     LDY #0
  400 .L5 LDA (addrlo),Y ;get length of
data item
  410     DEX            ;is this the ri
ght one??
  420     BEQ L6         ;yes - go and d
isplay it
  430     CLC            ;no - so point
to next item
  440     ADC addrlo
  450     STA addrlo
  460     BCC L5         ;loop back
  470     INC addrhi
  480     BNE L5         ;loop back
  490 .L6 TAY            ;YR=no. of byte
s to print+1
  500     DEY
  510 .L7 LDA (addrlo),Y ;get it...
  520     DEY
  530     STA (scrnlo),Y ;...and print i
t
  540     CPY #0         ;finished??
  550      BNE L7          ;no - back for
more
  560      RTS:]
  570 NEXT
  580 END
```

How it works

If you have followed the rather detailed explanations of what this program does, you should have a fairly good idea of how it goes about doing it. The program breaks down nicely into three parts:

Lines 90 – 320. The value of HIMEM + 8*X + 640*Y is worked out and stored (in binary) in locations scrnlo and scrnhi.

Lines 330 – 480. The shape to be displayed is located in the table and the address of its first byte is set up in locations addrlo and addrhi.

Lines 490 – 550. The shape is built up on the screen by transferring bytes from the table into screen RAM.

One point of interest is the way in which the *640 calculation is done. An early version of the routine actually did the multiplication, which is not too bad as 640 is a particularly easy number to multiply by in machine code (it's equal to 512+128). However, there is a 640 times-table held in the operating system and it seems a pity not to use it. This version of the routine is the one given here and owners of O.S.1.2 will find the table at &C375. It is well worth looking at this area with your Memory Display utility. If you have a different operating system you will have to find this table, so get out your String Search routine (this is what utilities are for!) and look for either &00008002 or &00000280 – the start of the table can appear in either form depending on whether the high or low bytes are stored first.

Surprisingly, the multiplication method only required 6 more bytes of store than the look-up technique used above; even so, that does represent an inefficiency in use of both time and storage. For interest the code is given here without comment – replace lines 120 – 160:

```
120     STA scrnlo
125     TYA
130     LDY #0
135     LSR A
140     BCC L1
145     LDY #128
150     CLC
155.L1  ADC scrnlo
160     ADC #&30
165     STY scrnlo
```

Extensions

This is one of those rare programs that really is finished: there is little left to do to it. If you intend to use it with your own machine code programs you will probably want to change the assembly address at line 70. Similarly, you are not obliged to store your table of data bytes at &2C00 – that was only a suggestion – and you may like to store the table in, or at the end of, your program. There is no real limit on the size of this table and it is determined only by the DIM statement in the previous routine. If you want more aliens/shapes then by all means extend it.

Two possible improvements which really deserve the description of 'rewrites' are to allow for fine vertical movement (by dots) and to handle shapes spanning two or three screen lines. Unless all of your characters span several lines it is probably best to deal with the latter situation by including the above routine and to call it once for each line to be output.

The first problem is far more difficult and (for starters) requires an extra parameter in the range 0 – 7 to define the start position within a character; we have assumed 0 in this routine, that is, the shape always starts at the top of a screen line. There are quite a few other complications besides so let us agree that we shall leave this one for the very dedicated among you! The routine given here is still valid up to the very last section (L7) and it is this coding that will have to be extensively rewritten.

You may also have noticed a slight reduction in horizontal resolution too, for our characters can only occupy 80 positions whereas the width of the screen is 160 dots. The complications involved in getting round this are enormous and simply not worth the effort since the difference in resolution appears minimal, even on a monitor.

Utility 23:

Graphics aid

Description

It is generally agreed that the BBC computer offers excellent graphics facilities, although they can sometimes be difficult to use. Essentially, all of the graphics commands are (or can be) VDU commands and a sequence of VDU codes may be used to draw any shape, however complex. This looks great in listings, but it can be difficult to read and – more importantly, from our point of view – tedious to write.

Suppose you want to draw a red rectangle, of some size, halfway down the right-hand side of the screen – what should you do? What often happens, mostly due to a lack of planning, is a time-consuming process of trial and error. Rather than get out the squared paper and the eight times tables it seems easier to draw any old red rectangle, check it, draw it again (bit closer that time) and again . . . by the third or fourth attempt, the result may be passable. Even if the figure is designed carefully beforehand, it is often necessary to see a shape on the screen to know if it is correct, however it looks on paper. Again, not everyone is expert on such things as 'relative plotting', 'colour masks', 'fill with logical inverse colour', etc., and some means of experimenting with these new ideas should be welcome.

This routine is halfway between being a utility and a fun program (by definition, utilities are not fun). It is a greatly extended version of the 'etch-a-sketch' type of program that allows you to draw on the screen using simple controls. In this routine we include a number of useful (and some quite advanced) plotting techniques and a status line that tells you what you are doing at any time. In the example of the red rectangle quoted earlier, you would draw the shape using the routines drawing facilities and then read off the GCOLs and co-ordinates that defined the shape. When you have finished the screen you would then be able to save it using the Screen Save utility given elsewhere in this section, print it out, or simply stand back and admire it.

Having chosen a MODE, the top line of the screen is reserved as a status line containing useful information, while the rest is defined as graphics window where you can play around to your heart's content. The basic actions consist of moving the cursor (a small dot) and selecting various options by pressing the function keys. Some of the facilities provided on the function keys are essential, while others are really a matter of personal choice. Those we have provided vary from the necessary 'join two points' to the frivolous 'Moiré mode'. All ten function keys are used.

Unless you are using this to doodle (yes, it's great for messing about on too!) you may need to refer to the status line to see what is going on. This line consists of seven fields and looks like this:

```
808 66 F 1 3 3 M
```

Notice that this is only given as an example. The significance of each field (numbering them 1 – 7 from the left) is as follows:

1) Cursor horizontal position. This is the horizontal displacement of the dot-cursor from the left hand edge of the screen. Its range is 0 – 1278.

2) Cursor vertical position. This is the vertical displacement of the dot-cursor from the bottom edge of the screen. Its range is 0 – 982 (slightly less than the maximum 1023 as the top line of the screen is reserved). Each of these co-ordinates is measured in suitable units for subsequent PLOT and DRAW commands.

3) Cursor speed. F stands for 'fast' and S stands for 'slow'. The S position is very useful for fine movement and high definition, while F is handy for zooming around the screen quickly.

4) Number of fixed points. You are allowed to 'fix' up to two points and this displays the number fixed at any time.

5) GCOL mode. The foreground colour (which is used for all your drawing) is indicated by the colour of this character, while the number indicates which GCOL effect is being used. For foreground plotting, the official range is 0 to 4 – see description in the *User Guide*.

6) Palette change. This field is used during a palette change to indicate the 'from' and 'to' colours. At any time it shows the result of the latest palette change.

7) Moiré mode. An M in this field indicates that the plotting is being done in a special way, loosely related to moiré patterns. Otherwise the field will be blank.

In the next section we look at the various features available.

Use

The program is in BASIC and can be RUN. It is short enough to be able to operate on a disc machine and in all MODES, so it should run on any (32K) system.

Once you have selected the MODE, the function keys come into effect and we shall look at those shortly. The only non-function keys that are relevant are the cursor keys, Q and DELETE. The cursor keys are used to move the cursor around – the movement features full wrap-around and auto-repeat. DELETE will delete the last fixed point if there was one, otherwise it will do nothing. This is necessary as certain functions set their own fixed points which you may not require. Finally, Q is used to quit the program, and to reset the cursor and function keys to their normal modes of operation.

You can press a function key at (almost) any time to select an effect – these are now described in detail.

f0: Speed select. This key toggles the speed setting between F and S.

f1: Fix point. If you have not used both fixed points, this will fix one for you. The point is left behind when you move the cursor away and will appear in the current foreground colour.

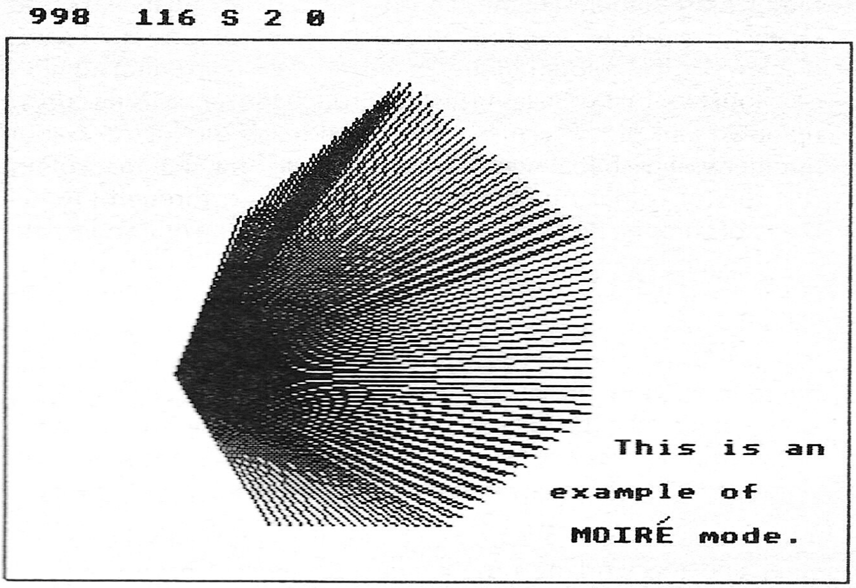
f2: Join. Joins the previous fixed point to the current cursor position with a straight line. Notice that, if there are two fixed points when you do the join, the first will be lost and the current cursor position is inserted at the top of the list of fixed points. This means that you can move around and 'join' to create a polygon without having to specifically save any of the vertices – that is done for you. If you do not want to remember the last point joined as a fixed point, use DELETE to get rid of it.

f3: Triangle. Draws a (filled) triangle using the last two fixed points and the current cursor position as vertices. This implies, of course, that you must have two points already fixed before you can draw the triangle. If not, the request is ignored. The current cursor position is saved as the latest fixed point and the one saved before that will also be remembered. Consequently, after a 'triangle' command, there will still be two fixed points (as there were before), but they will be the last two points visited. This scheme makes it easy to draw rectangles and other shapes composed of triangles because it has a chaining effect analogous to that of the 'join' command.

f4: Circle. Draws a (filled) circle centred at the latest fixed point so that the current position lies on the circumference. The centre is held as the last fixed point so that a series of concentric circles is easily drawn. If there are no fixed points the request is ignored.

f5: Moiré. Selects moiré mode – the key acts as a toggle between moiré on and off. As the cursor moves, straight lines are continually drawn to the last fixed point. Because of the relatively low resolution, even in MODE 0, the slight imperfections in the lines give the appearance of a moiré pattern. Strictly speaking, a moiré pattern consists of two almost identical patterns overlayed to give 'interference' effects, but the results in this mode are very similar.

An example of the display in this mode is reproduced below – this printout was done using the Printer Dump utility given in Section 4. In this diagram, the frame around the graphics area was drawn using 'join'; normally, no such boundary is used unless you draw one yourself.



```
998   116 S 2 0
```

This is an

example of

MOIRÉ mode.

f6: Text. Allows you to type in text to label diagrams and plans etc. The cursor position will be the top left hand corner of the first text character and the colour of the print is the same as the current foreground colour. To exit from this mode, press <RETURN>. Notice that DELETE does not work in this mode, neither can you erase letters by overwriting them, as the printing is done using VDU 5, which means that characters will overlay others rather than wipe them out. This is actually a rather useful feature as it enables you to produce different styles of lettering.

f7: Change colour. The current GCOL colour can be changed by holding down either the 'cursor up' or 'cursor down' key to step through the available colours. To indicate that you are in this mode, field 5 is replaced by a solid block whose colour indicates the colour you will select by pressing <RETURN>. All subsequent plots will use this colour. When <RETURN> is pressed, the colour is retained, but the field is replaced by the current GCOL mode number.

f8: Change GCOL. The cursor up/down keys will step the GCOL mode through the values 0 – 4. Leave this mode by pressing <RETURN>. Initially this field is set to 0 which means that all plotting will be done in the colour specified, i.e. white.

f9: Palette. In MODEs other than 2, you might like a rest from the rather drab colour scheme, and use of this key enables you to change the palette, thereby introducing new colours. Field 6 will display a logical colour number in the range 0 – (maximum number of colours for that MODE less 1); step through the range using cursor up/down and press <RETURN> when you reach the logical colour number you wish to amend. 'Cursor up' will now step that colour (the logical colour number will change colour) through the full range of colours. Press <RETURN> when you find the one you want.

This sounds very complicated but in practice it is easy to use. For example, if we were in MODE 1, colour 2 would normally appear yellow. To change this to say, magenta, we would select the 'palette' facility and hold down a cursor key until field 6 showed the number '2'. Now press <RETURN>. Holding down 'cursor up' will swap colour 2 for the full paint-box of colours – stop when it turns magenta and press <RETURN>. Notice that anything that appears in yellow will change colour in sympathy with the swapping colours.

These facilities provide a good range of material for both the experimenter and the working programmer. Recently, I have used this routine to design a Backgammon board and the task was greatly simplified by the ability to change palettes and to read off the X and Y co-ordinates of key points once the board began to take shape. On the other hand, it can provide some interesting information about the effects of different GCOL setting and colour changes.

The routine can easily be amended to accommodate other graphic effects, depending on your own requirements. Some suggestions are given below.

```
   10 REM GRAPHICS AID
   20 REPEAT
   30 INPUT "MODE (0,1,2,4,5) "M%
   40 mc%=VALMID$("020416000204",2*M%+1,
2)
   50 UNTIL mc%>0
   60 MODE M%
```

```
 70 DIM FX%(2),FY%(2)
 80 mx%=1280:my%=984
 90 @%=4:X%=0:Y%=0:D%=16:nf%=0
100 gcol%=0:was%=0:fix%=0:moire%=0
110 white%=(mc%-1) AND 7
120 col%=white%
130 VDU 23,1,0;0;0;0;
140 VDU 24,0;0;mx%-1;my%-1;
150 VDU 23,224,-1,-1,-1,-1,-1,-1,-1,-1
160 PRINT"   0    0 F 0 0"
170 GCOL 0,7:PLOT 69,0,0
180 *FX 4,1
190 *FX 225,200
200
210 REPEAT
220 F%=INKEY(0)
230 IF F%=200 PROCspeed
240 IF F%=201 AND nf%<2 PROCfix
250 IF F%=202 AND nf%>0 PROCjoin
260 IF F%=203 AND nf%>1 PROCtring
270 IF F%=204 AND nf%>0 PROCcircle
280 IF F%=205 PROCmoire
290 IF F%=206 PROCtext
300 IF F%=207 PROCcolour
310 IF F%=208 PROCgcol
320 IF F%=209 PROCpalette
330 IF F%=127 AND nf%>0 PROCdel
340 IF F%=136 PROCmove(-D%,0)
350 IF F%=137 PROCmove(+D%,0)
360 IF F%=138 PROCmove(0,-D%)
370 IF F%=139 PROCmove(0,+D%)
380 UNTIL F%=81
390 *FX 4,0
400 *FX 225,1
410 END
420
430 DEFPROCspeed
440 PROCbeep(6)
450 D%=18-D%
460 IF D%=2 A$="S" ELSE A$="F"
470 PRINT TAB(10,0)A$
480 ENDPROC
490
500 DEFPROCfix
```

```
  510 PROCbeep(7)
  520 @%=1:fix%=1
  530 nf%=nf%+1
  540 FX%(nf%)=X%:FY%(nf%)=Y%
  550 PRINT TAB(12,0)nf%
  560 ENDPROC
  570
  580 DEFPROCjoin
  590 PROCbeep(8)
  600 GCOL gcol%,col%
  610 DRAW FX%(nf%),FY%(nf%)
  620 FX%(nf%)=X%:FY%(nf%)=Y%
  630 ENDPROC
  640
  650 DEFPROCtring
  660 PROCbeep(9)
  670 MOVE FX%(1),FY%(1)
  680 MOVE FX%(2),FY%(2)
  690 GCOL gcol%,col%
  700 PLOT 85,X%,Y%
  710 FX%(1)=FX%(2):FY%(1)=FY%(2)
  720 FX%(2)=X%:FY%(2)=Y%
  730 ENDPROC
  740
  750 DEFPROCcircle
  760 PROCbeep(10)
  770 R=SQR((FX%(nf%)-X%)^2+(FY%(nf%)-Y%
)^2)
  780 VDU 29,FX%(nf%);FY%(nf%);
  790 S=PI/30:MOVE R,0
  800 GCOL gcol%,col%
  810 FOR P=S TO 2*PI STEP S
  820 MOVE 0,0
  830 PLOT 85,R*COSP,R*SINP
  840 NEXT
  850 VDU 29,0;0;
  860 ENDPROC
  870
  880 DEFPROCmoire
  890 PROCbeep(11)
  900 moire%=1-moire%
  910 IF moire% A$="M" ELSE A$=" "
  920 PRINT TAB(19,0)A$
  930 ENDPROC
```

```
 940
 950 DEFPROCtext
 960 PROCbeep(12)
 970 VDU 5
 980 REPEAT
 990 W%=GET
1000 IF W%>31 AND W%<127 VDU W%
1010 UNTIL W%=13
1020 VDU 4,23,1,0;0;0;0;
1030 ENDPROC
1040
1050 DEFPROCcolour
1060 PROCbeep(13)
1070 @%=1
1080 REPEAT
1090 F%=INKEY(0)
1100 IF F%=138 PROCnewcol(-1)
1110 IF F%=139 PROCnewcol(+1)
1120 UNTIL F%=13
1130 PRINT TAB(14,0)gcol%
1140 COLOUR white%
1150 ENDPROC
1160
1170 DEFPROCnewcol(dy%)
1180 SOUND 1,-9,240,1
1190 col%=(col%+dy%+mc%) MOD mc%
1200 COLOUR col%
1210 PRINT TAB(14,0)CHR$224
1220 ENDPROC
1230
1240 DEFPROCgcol
1250 PROCbeep(15)
1260 COLOUR col%
1270 REPEAT
1280 F%=INKEY(0)
1290 IF F%=138 PROCnewgco(-1)
1300 IF F%=139 PROCnewgco(+1)
1310 UNTIL F%=13
1320 COLOUR white%
1330 ENDPROC
1340
1350 DEFPROCnewgco(dy%)
1360 gcol%=(gcol%+dy%+5) MOD 5
1370 @%=1:PRINT TAB(14,0)gcol%
```

```
1380 ENDPROC
1390
1400 DEFPROCpalette
1410 PROCbeep(16)
1420 @%=2:V%=0
1430 REPEAT
1440 F%=INKEY(0)
1450 IF F%=138 PROCchcol(-1)
1460 IF F%=139 PROCchcol(+1)
1470 UNTIL F%=13
1480 VDU 7
1490 REPEAT
1500 F%=INKEY(0)
1510 IF F%=139 PROCvducol
1520 UNTIL F%=13
1530 COLOUR white%
1540 ENDPROC
1550
1560 DEFPROCchcol(dy%)
1570 V%=(V%+dy%+mc%) MOD mc%
1580 PRINT TAB(16,0)V%
1590 ENDPROC
1600
1610 DEFPROCvducol
1620 W%=(W%+1)MOD 16
1630 VDU 19,V%,W%,0,0,0
1640 COLOUR V%:PRINT TAB(16,0)V%
1650 ENDPROC
1660
1670 DEFPROCdel
1680 PROCbeep(3)
1690 GCOL 0,0
1700 PLOT 69,FX%(nf%),FY%(nf%)
1710 nf%=nf%-1
1720 @%=1
1730 PRINT TAB(12,0)nf%
1740 ENDPROC
1750
1760 DEFPROCmove(dx%,dy%)
1770 IF fix%=0 GCOL 0,was%:PLOT 69,X%,Y
%
1780 X%=(X%+dx%+mx%) MOD mx%
1790 Y%=(Y%+dy%+my%) MOD my%
1800 was%=POINT(X%,Y%)
```

```
1810 GCOL 0,col%:PLOT 70,X%,Y%
1820 @%=4:fix%=0
1830 PRINT TAB(0,0)X%" "Y%
1840 GCOL gcol%,col%
1850 IF moire% DRAW FX%(nf%),FY%(nf%)
1860 ENDPROC
1870
1880 DEFPROCbeep(W%)
1890 SOUND 1,-9,12*W%,10
1900 ENDPROC
```

How it works

Once the initialisation has been completed, the program enters a large loop (lines 210 – 380) which checks for any valid key being pressed. Action is taken depending on the key found and, with the exception of Q (quit program) this causes a PROCedure to be called to handle the processing of that request. On completing the PROCedure, control is returned to the loop.

Procedures

The bulk of this program is made up of PROCedures – for a start, each key option has an associated PROCedure. These are:

Key	PROCedure Name
f0	speed
f1	fix
f2	join
f3	tring
f4	circle
f5	moire
f6	text
f7	colour
f8	gcol
f9	palette

The function of each of these PROCedures should be obvious – the set is completed by some less obvious ones:

PROCnewcol is called by PROCcolour to step the foreground colour through a range of values. The parameter indicates whether the step is up (+1) or down (−1). Similarly, PROCnewgco performs the same service for PROCedure PROCgcol. Two such PROCedures are required by the palette-changing PROCedure; one to change the logical colour number and one to change the actual colour once the logical colour has been selected. These are denoted by PROCchcol and PROCvducol respectively.

If there are fixed points, pressing the DELETE key will remove the latest one and calls PROCdel to do so.

One very important routine is PROCmove which is used to move the cursor, update the display and draw a new line if 'Moiré' mode is set; this PROCedure is called each time a cursor key is pressed. It takes two parameters, namely: amount of horizontal and amount of vertical movement in that order.

Last and least, PROCbeep does just that.

Variables

Variables abound in this program – here is a list of the important ones, together with their meanings.

FX%(2)	The X co-ordinates of the fixed points.
FY%(2)	The Y co-ordinates of the fixed points.
mx%	The width of the screen (constant = 1280).
my%	The height of the screen (constant = 984).
X%	The current X co-ordinate of the cursor.
Y%	The current Y co-ordinate of the cursor.
D%	Displacement of each step in a move. For 'slow' D%=2 and for 'fast' D%=16.
nf%	The number of fixed points at any time.
gcol%	The current GCOL mode in the range 0 – 4.
col%	The current foreground (plotting) colour.
was%	The colour of the point 'underneath' the cursor.
fix%	Set to 1 to indicate that the point beneath the cursor is to be replotted, instead of being blanked out. Otherwise fix% is set to 0.
white%	The number that represents white in the current MODE. With one exception (what is it?), this is one less than the number of colours available in that MODE.

In addition quite a few 'local' variables are used within PROCedures (but not across the program) as temporary storage. The use of these variables is reasonably obvious when you look at how they are used. One variable may serve several functions in the different sections in which it appears, but because of the way it is used, this will not cause any problems.

Extensions

The number of variations is basically limited to the number of functions you can squeeze out of your function keys. In this utility, the keys are programmed to generate ASCII codes from 200 upwards. To get more from them, you can reprogram the CTRL, SHIFT and CTRL/SHIFT versions of the key (using *FX 225 – 228) and extend the program loop by including the relevant procedures.

As written, the routine falls neatly between 'useful' and 'enjoyable' – if you have more (or less) serious uses for it, then this will be reflected in the modifications you choose to include. For example, if you want to use the program for drafting out plans or mathematical figures a grid of squares would be useful for guidance. One key could be used to switch the grid in or out.

Another useful facility is sometimes known as 'rubber-banding' which means that as you move a line around, the previous copy of the the line is deleted giving the illusion of a rubber band fixed at one point with the other being dragged around the screen under your control. This is an easy amendment, as the 'Moiré' facility is itself a simplified version of this technique. To enable the 'Moiré' mode to become a 'rubber-band' mode you will need the following additions to the routine:

```
 1765IF moire% MOVE X%,Y%: GOTO 1780
 1795IF moire% GCOL 3,col%: DRAW FX%(nf%
),FY%(nf%)
```

The effect of this is to EOR the colour of the line you have just drawn with itself, producing black. (Probably – it depends very much on how the colours got onto the screen in the first place. Because this is not a 'pure' effect, and depends on other factors, it has been omitted from the original list of facilities.)

Unless you use a monitor for your display, you have some difficulty in reading MODE 0 screens. At present, the cursor is only the size of the smallest dot possible in the chosen MODE and this will not be visible on a television set. One worthwhile improvement would be to enlarge the cursor – perhaps to a 'crosshair sight' – so that it is easier to see. You might like to do this for yourself by amending lines 1770 – 1810.

We have already observed that any series of graphics commands can be thought of as a string of VDU statements. In many cases, the instructions that create a screen are far more compact than the screen itself and that set of instructions is nothing more than a list of VDU codes. This suggests the interesting possibility of remembering significant codes (ones that contribute to the form of the screen, rather than those associated with your wandering around it) and saving them for future use.

As an example, the four bytes:

12, 17, 1, 65

are easily remembered and, when preceded by VDU, they will clear a MODE 2 (say) screen and print a red letter A in the top corner. This may not seem very interesting, but if you want to store that screen (in the normal way with *SAVE), it will cost you 20,480 bytes! The four single byte codes, together with a small overhead to do the VDUing are definitely a better bet. Even fairly complicated screens can be broken down into a shortish list of VDU codes which could be stored and then 'drawn' back onto the screen using VDU commands (or OSWRCH if you are writing in Assembler).

To implement this feature in the utility will require some care. Basically, an area must be set aside to hold the data and important VDU codes (such as those from PLOT or GCOL commands) should be stored there as they are executed. When the screen is complete, saving the table of data is tantamount to saving the instructions for drawing the screen and those instructions can be implemented by a one-liner as simple as this:

```
MODE 2:X=OPENIN("data"):REPEAT:
VDU BGET#X:UNTIL EOF#X
```

This interesting technique would allow many 'screens' to be stored in a much smaller space than would normally be occupied by just one normal high-resolution screen. Furthermore, for certain types of screen, it would be much faster to 'draw' it than it would be to *LOAD it from tape (but probably not disc); and so, for once, we gain on the roundabouts *and* the swings.

Although this feature is not included in our utility, the program was written with the idea in mind and the amendments should not be too difficult. The best approach would be to take one function at a time and to include in its PROCedure the correct instructions for generating the appropriate VDU codes. Calling a new PROCedure would execute that list of codes (thereby updating the screen) and also store them in the table. Before you quit the program, *SAVE the table to tape or disc.

If you go through with this modification you will need one further PROCedure to read the data back into the program, ready for further processing. We have already seen that this is a fairly simple job. Alternatively, the code could be merged into an entirely different program, for example a game. This could then create the screen from a small amount of data held either within the program (as DATA) or on file outside it. If you decided on the latter approach, then the data would be available to any program that cared to use it, provided it contained the 'drawing' procedure.

Although this routine is more 'experimental' than some of our others it does, nonetheless, contain a number of useful features and will be found to be extremely helpful in the design of graphics effects. Effectively it allows you to throw away your sketch-pad and graph paper and to create your graphics where they are most at home – on the screen.

Utility 24:

MODE 7 pixel plotter

Description

Many programs for the BBC computer occupy so much memory that they have to use the teletext mode (MODE 7) as this MODE only requires 1K of RAM for its screen. This immediately imposes limits on graphics since MODE 7 only supports 80 by 75 resolution, although it does allow all eight colours and certain other effects. Provided a program does not require very high resolution, this is a good MODE to work in and it is very often used for display screens in game and educational programs; perhaps the best example of this is the famous 'ACORNSOFT' screen – surely familiar to all owners of a BBC Micro. Despite the lack of resolution, this is an effective screen and a good example of what can be achieved with some effort.

The appendix to the *User Guide* shows the characters available in MODE 7; quite a number of these (128 – 159) are control codes which determine the appearance of the rest of the characters on the line. The remaining characters include some other control codes (0 – 31), the usual teletext character set and a collection of graphics shapes based on a 2 by 3 grid and the same size as a normal text character.

Experience has shown that getting these characters onto the screen involves a fair bit of trial and error even if elaborate plans are made beforehand. This utility lets you create MODE 7 screens in a more versatile way by enabling you to plot the individual pixels which make up the graphics characters. So that complete teletext screens may be created, it also enables you to enter both control codes and text and so you can achieve virtually any effect possible with a MODE 7 screen. Once the screen has been created, it is a good idea to SAVE it in some form or other and the utilities 'MODE 7 procedure writer' and 'Screen save' should help you here. Essentially, this routine provides a MODE 7 'doodle' facility in the same way that the 'Graphics aid' utility does for the other graphics modes.

The cursor is moved round the screen in pixel-sized steps – that is, on an 80 by 75 grid, by using the cursor keys. To plot or unplot a Pixel, press P and to plot a full 2 by 3 Block, press B. Control codes can be set up by using CTRL and/or SHIFT in conjunction with the function keys, and text can be entered once Text mode is established by pressing T. The full list of available facilities is given in the next section.

Even if you have avoided graphics effects in MODE 7 because of the many complications, you will find that this utility enables you to produce professional-looking screens quickly and with much less effort than would normally be required. As with the 'Graphics aid', the fact that you can see the screen as you build it up means that you can expect to be right the first time, rather than the 99th.

Use

This routine can be RUN by itself or may be merged with other routines, two of which have already been mentioned. Let us take the simplest case, where the routine is RUN on its own.

MODE 7 is selected and a rapidly flashing pixel-sized cursor is located in the top left hand corner. The following keys are now effective:

Cursor keys: Used to move the cursor around the screen (with auto repeat and wraparound) in both text and graphics modes.

B: This fills in the whole of the character at the current character position. (B stands for 'Block').

F: Prints a 'flashing' control code at the current cursor position and steps the cursor to the right.

S: As above, except that the 'steady' (not flashing) control code is sent to the screen.

P: Plots or unplots the pixel at the current cursor position. The cursor does not move once the pixel has been plotted and you can only really see the effect by using one of the movement keys to step the cursor away. This is the most important function of the utility.

Q: Quit the program. The cursor is returned to the bottom line of the screen so that you may admire your creative efforts. If you use this utility in conjunction with another to save the screen, it should be done on receipt of the Q key.

CTRL–fn: The function keys generate control codes in the range 144 – 153 when used in conjunction with the CTRL key. These control codes are used to define the attributes of the following graphics characters (see *User Guide* appendix). For example, CTRL–f3 generates a 'graphic yellow' control code and subsequent blocks and pixels (on that line) will appear in yellow until you send a new colour code.

CTRL–SHIFT–fn: Pressing CTRL and SHIFT together with a function key will generate control codes in the range 154 – 159 which, along with those mentioned above, enable all of the graphics effects described in the *User Guide* to be realised.

T: Switches into text mode. This time, typing a letter at the keyboard will print that character on the screen. The usual cursor is restored and can be moved using the cursor control keys. While you are in this mode, it is not possible to send graphics control characters to the screen, but you can send text control codes by using a SHIFT–fn combination, these are the usual codes as sent by these keys (for example, SHIFT–f1 gives 'alpha red'). While you are in this mode you cannot send the flashing and steady codes – they can only be sent from the pixel mode. To leave text mode, press CTRL–X.

As you move the cursor around the screen it may change its form, depending on the character at the location it has just moved to. If the cell is blank, or occupied by a graphic character, the usual pixel cursor appears, whereas if the character is a text item or control code the familiar underline cursor is used.

It is the control codes that determine how the remaining characters on the screen line are to be interpreted and you need to know where they are. To help you spot them, the machine sounds a warning when you land on a control code – you may overwrite it with another character, but you will probably want to move off it as soon as possible. Notice that if you are travelling vertically at the time, you will need three steps (pixel-sized, if you remember) to escape from it. Perhaps I shouldn't say this in a book of serious programs (such are utilities) but there has to be the basis of a good game here.

```
   10 REM MODE 7 PIXEL PLOTTER
   20 ON ERROR GOTO 330
   30 MODE 7
   40 DIM A%(1,2)
   50 A%(0,0)=1:A%(0,1)=4:A%(0,2)=16
   60 A%(1,0)=2:A%(1,1)=8:A%(1,2)=64
   70 ENVELOPE 1,2,255,6,-1,1,4,2,90,-5,
-4,-60,126,99
   80 *FX 4,1
   90 *FX 12,1
  100 *FX 228,154
  110 FOR I%=0 TO 24
  120 VDU 31,0,I%,151
  130 NEXT
  140 X%=2:Y%=0
  150 PROCmove(0,0)
  160
  170 REPEAT
  180 F%=INKEY(0)
  190 IF F%=66    sq%=255
  200 IF F%=80    sq%=sq% EOR byte%
  210 IF F%=84    PROCtext
  220 IF F%=136   PROCmove(-1,0)
  230 IF F%=137   PROCmove(+1,0)
  240 IF F%=138   PROCmove(0,+1)
  250 IF F%=139   PROCmove(0,-1)
  260 IF F%=70    F%=136
  270 IF F%=83    F%=137
  280 IF F%>135   AND F%<160 PROCccode
  290 IF sq%>159 PROCflash
  300 IF F%>=0 THEN *FX 15,1
  310 UNTIL F%=81
  320
  330 ON ERROR OFF
  340 *FX 4,0
  350 *FX 12,0
  360 *FX 228,128
  370 VDU 31,0,24,23,1,1;0;0;0;
  380 END
  390
  400 DEFPROCmove(dx%,dy%)
  410 F%=0
  420 X%=(X%+80+dx%) MOD 80
  430 Y%=(Y%+75+dy%) MOD 75
  440 byte%=A%(X% MOD 2,Y% MOD 3)
  450 x%=X% DIV 2:y%=Y% DIV 3
  460 VDU 31,x%,y%
  470 A%=&87
  480 sq%=(USR(&FFF4) AND &FFFF) DIV &10
0
  490 IF sq%=32 sq%=160
  500 IF sq%<160 V%=1 ELSE V%=0
  510 IF sq%>128 AND sq%<160 SOUND 1,1,1
00,12
  520 VDU 23,1,V%;0;0;0;
  530 ENDPROC
  540
  550 DEFPROCccode
  560 SOUND 1,-12,200,5
  570 IF sq%<128 OR sq%>159 VDU 8
  580 VDU F%
  590 PROCmove(2,0)
  600 ENDPROC
  610
  620 DEFPROCflash
```

```
630 T%=TIME
640 VDU 31,x%,y%,sq% EOR byte%
650 REPEAT UNTIL TIME>T%+2
660 VDU 31,x%,y%,sq%
670 REPEAT UNTIL TIME>T%+4
680 ENDPROC
690
700 DEFPROCtext
710 *FX 12,0
720 VDU 8,23,1,1;0;0;0;
730 REPEAT
740 F%=INKEY(0)
750 IF F%>135 AND F%<140 VDU F%-128
760 IF F%>31  AND F%<136 VDU F%
770 UNTIL F%=24
780 *FX 12,1
790 VDU 23,1,0;0;0;0;
800 X%=2*POS:Y%=3*VPOS
810 PROCmove(0,0)
820 ENDPROC
```

How it works

As you move to a new cell (character position) on the screen, the previous contents of that cell are read and remembered. If the previous occupant was a text character or a control code then the standard cursor is printed and a warning is sounded if the character was a control code. This action is not very interesting – it is when the cell turns out to contain a graphic or space character that the real work starts. When this happens the exact position of the cursor becomes important and the relevant pixel is flashed. This flashing is done by the program as part of its loop; it is not automatic.

For graphics, each cell is split into six separate pixels and the relationship between the pixels and the character code is like this:

Code = 160 +

1	2
4	8
16	64

If you study the *User Guide* appendix you will see that a few codes less than 160 also generate some of these shapes, although the relationship between codes and shapes is not consistent if the codes are less than 160. In fact, one of the first things the routine does entering a 'blank' cell, that is, one containing a space character (ASCII code 32), is to replace the code by 160 so that it is thought of as a graphic character on all future visits.

As the cursor moves round the screen, the position of its current cell is remembered and so too is the position of the pixel cursor *within* that cell –it can occupy one of six locations as shown above. The corresponding code for this position is called byte% and it is used to flash the pixel and also to plot it, when this is required. The use of byte% is greatly simplified by use of the BASIC verb EOR – as you should be aware, this operation is its own inverse; in other words, if you want to cancel the effect of an EOR operation you can simply repeat the operation and it will cancel itself out. Both program lines (200 and 640) that feature byte% use this verb.

The program itself consists of a loop which continually scans the keyboard, waiting for a suitable key to be pressed. Pressing T calls a PROCedure specifically for dealing with text input, as this is rather different from the graphics mode. Similarly, movement and function keys are handled by their own PROCedures.

At the end of the loop a flash is performed if the cursor is found to be at a graphic character's cell. When Q is pressed, the loop terminates and the cursor and function keys are reset before END.

The main purpose of this utility is to help you plot individual pixels and their associated graphics codes. To get us off to a good start, the program writes a 151 code (graphic white) at the start of each line which ensures that you start in graphics mode. If you want to put some text on say, the top line, then you should delete this code from the start of the line. When the (pixel) cursor first appears, moving left will start the alarm as you have moved onto a control code and the real cursor will appear. Now select text mode and overwrite the code with one your own choice.

Procedures

To move the cursor around the screen, PROCmove is called with two parameters; namely the amounts of horizontal and vertical movement required. As the move is made, the status of the moved-to cell is calculated and so using two zero parameters enables us to calculate the status of the current cell and pixel position without actually moving from it. If it turns out that we have moved onto a control code, the alarm is sounded.

If you decide to type in a control code, then PROCccode is called to print the code for you and to move the cursor to the next cell.

Each time the program loop reaches the end, a check is performed to see if the cursor is on a graphics character. If so, then the pixel must be displayed and PROCflash is used to do so. This simple procedure is coded in such a way that the contents of the cell are left unchanged by it – that is, the current pixel is first swapped and then replaced after a short pause. To keep the program moving, it is not possible to dwell on this wait for too long so the 'flash' rate is actually rather fast, although it is quite easy to tell which pixel is being flashed.

Finally, PROCtext is entered in response to a 'T' command. As the text mode is rather different from graphics, it merits its own PROCedure, making only brief forays to PROCmove to read the contents of the cell it is about to enter.

Variables

We have seen that each pixel in a block is associated with a power of 2 that is used to compose the ASCII code for the shape. Unfortunately these powers are not consecutive – 32 is missing – and so it is not so easy to convert 'position within pixel' to 'power of 2'. To simplify this job we use a small array A%(1,2) as a look-up table; the six members of this array are defined in lines 50 and 60 and are maintained throughout the program.

The position of the cursor is held in two forms: X% and Y% are based on an 80 by 75 grid and contain the location of the 'pixel cursor' while x% and y% refer to actual screen co-ordinates and are based on a 40 by 25 grid. For all movement, X% and Y% are updated and x% and y% are calculated from them.

The use of byte% has already been seen – during graphics mode it will take one of the values 1, 2, 4, 8, 16 or 64. It is necessary to remember the previous value of the cell so that the pixel may be plotted or removed (or flashed); this value is stored in sq% and is updated each time a new cell is entered or the current one is edited. The decision as to whether we have moved onto a control code or graphics code or text character is based on sq%.

Extensions

You can use this routine to doodle on the screen but you will probably want to SAVE the results and so it should be used together with a SAVE routine. A general SAVE routine is given earlier, but for this very specialised job a much simpler method can be used:

```
315 *SAVE "Screen" 7C00 +3FF
```

will save the screen before quitting the program and:

```
*LOAD "Screen"
```

will get it back for you. This is not the only way of saving the screen – for example, you cannot use this method in the middle of a program without invoking the filing system and, in the case of tapes, this could mean quite a long wait. An alternative approach is to store the screen as a series of integers and to 'pling' it back from a BASIC program when you require it. This is slightly more complicated than the previous approach, although it is more versatile. This amendment will save the screen in a suitable form:

```
312 *OPT 1,0
314 X=OPENOUT("SCREEN")
316 FOR I%=0 TO &400 STEP 4
318 Y%=I%!&7C00
320 PRINT#X,Y%
322 NEXT
324 CLOSE#X
```

The coding required to get this data back into the computer is rather similar but is given here for completeness:

```
10 CLS
20 X=OPENIN("SCREEN")
30 FOR I%=0 TO &400 STEP 4
40 INPUT #X,Y%
50I%!&7C00=Y%
60 NEXT
70 CLOSE#X
```

Notice that this loads the data right back where it came from, that is, onto the screen. However, this is not necessary and you could, for example, pling it onto the end of your BASIC program (starting at TOP, say). Once it is there, this line will get it onto the screen for you:

`FOR I%=0 TO &400 STEP 4: I%!&7C00=I%! TOP:NEXT`

The advantage of this technique is that the screen (or several screens) can be summoned from within the program without needing to refer to the filing system. Quite a number of commercial programs use this method to create a display screen while another section is loading. The loader is a BASIC program that has been *SAVED so that it includes the data for plinging to the screen, probably saved at TOP.

As usual, the approach you adopt will depend very much on your intended application.

One improvement you might like to make is to introduce some new key facilities. As an example, 'I' could be used to Invert the state of each pixel within the current cell (a sort of 'reverse field' instruction) – this is dead easy; EOR it with 255–160=95 (why 95 ? – try thinking in binary). Similarly if you use any particular code a lot, it may be an idea to give it its own key rather than leave it as a combination of SHIFT and CTRL function key.

Section 4:
Miscellaneous utilities

In this section we look at a few assorted routines which are grouped together here primarily because they don't belong anywhere else. Two of these utilities apply specifically to peripherals, namely the Disc Sector Editor (DSE) and the Printer Screen Dump. Even if you do not use discs or a printer, you should find something of interest in the explanations of how these routines work and what they do. For example, it has already been suggested that you may wish to incorporate the DSE editing facilities in other programs. Similarly, the String Sort includes a method of organising files which is useful even if you have no need of data sorting.

Although this section contains the final selection of our utilities, there are many other possible routines to help you get the most out of your BBC computer. Throughout this book we have tried to encourage you to extend and improve the programs and have hopefully pointed the way to some interesting and worthwhile investigations. Soon it will be your turn to sweat over an ophidian algorithan, but first let us look at our final set of programs.

Utility 25:

Display user keys

Description

Most programmers have their own pet uses for the BBC's user-defineable keys. Some like to store BASIC keywords on them so that PRINT", DATA, or whatever are available at a touch, while others use them to store mini-programs, as with some of the utilities we have already seen. Without doubt they are one of the most practical features of the machine and during a serious hacking session will probably be re-programmed many times for different tasks.

Even if you physically label the keys with the initial definition assigned to them, you may well reset them in the course of a program and it is likely that, after a while, you will never be sure exactly what each key has been programmed to do. The simple answer is to press it and see, but that may have dire consequences. A safer way is to type a line number (0 is a good one) and then press the key so that its contents are printed out but not executed. This is sufficient for most cases but will not always present the full definition.

A more reliable method is to read the definition from the key storage area, which is how the next routine works. Using this utility we can see exactly how each key has been programmed since it displays the character strings currently assigned to all the function keys.

At this point we should remind ourselves that there are sixteen such keys (see the notes on *FX4 in the *User Guide*) and in addition to generating character strings they can also produce ASCII codes in conjunction with the SHIFT and CTRL keys. We have already seen this aspect of their use in some of our previous utilities. The ASCII code facility is independent of the character string associated with the key and does not concern us here.

To print out the key contents is not a difficult exercise and you might like to have a go at it yourself. The first step is to check out Page 11 (&B00 onwards) using our memory display utility. This is the area where the key definitions are held and with a little experimentation you should see the technique that Acorn have utilised to store them. We won't go into the details here, but the important features will be considered in the 'How it works' section.

When RUN the routine prints out the contents of all sixteen keys, surrounding the strings with a pair of '■' characters. The reason for this is that, if the string has been set up to print, say, eight spaces (for example, if you are writing neat assembler code), then the spaces will not show unless they are delimited by something. (Thanks to G.P. for this one.)

Use

The utility is RUN like any other BASIC program and will fit into three pages, so set PAGE=PAGE+&300 before LOADing. Obviously, once the keys have been listed to the screen they are fair game for editing using the COPY key. This is a useful facility if you have typed in a long, but not quite correct, key definition as you can then COPY it having RUN the utility. Also, if required you could string two or more definitions together on one key.

```
   10 REM DISPLAY USER KEYS
   20 MODE 7
   30 PRINT TAB(7)"CONTENTS OF FUNCTION
KEYS"'
   40 FOR K%=0 TO 15
   50 PRINT "KEY ";K%;"  ";CHR$255;
   60 PROCkey
   70 NEXT
   80 PRINT'
   90 END
  100
  110 DEFPROCkey
  120 ptr%=K%?&B00:max%=255
  130 FOR I%=0 TO 15
  140 disp%=I%?&B00
  150 IF I%=K% OR disp%<ptr% THEN 180
  160 IF disp%<max% max%=disp%
  170 IF disp%=ptr% I%=15:max%=255
  180 NEXT
  190 IF max%<255 PROClistkey
  200 PRINT CHR$255
  210 ENDPROC
  220
  230 DEFPROClistkey
  240 REPEAT
  250 ptr%=ptr%+1:byte%=ptr%?&B00
  260 IF COUNT=39 PRINT'"        ";
  270 IF byte%<32 PRINT "¦";:byte%=byte%
+64
  280 IF byte%>128 PRINT "¦!";
  290 PRINT CHR$byte%;
  300 UNTIL ptr%=max%
  310 ENDPROC
```

How it works

The key definitions are packed into Page 11 one after the other with no obvious gaps between them, so how does the computer know where each key definition starts and finishes?

The first part of this is easy. A table starting at &B00 points to the start of each definition and is updated each time a key is (re)defined. If you enter *KEY5 LIST then the entry at &B00+5 is updated to point to your string, the pointer being a single byte binary displacement from &B01. You will notice that all of the free keys have their pointers updated even though they apparently have nothing to point at. This can be checked by using your memory display utility.

So finding the string start is easy, but finding the end is something of a problem. If the keys are entered in strict numeric order then clearly a key definition ends as soon as the next one starts, but this situation is not likely to arise often. What we must do is to scan the displacements for all of the other keys and find the next highest above any new displacement. This is where the next key starts and, by inference, where the new one stops.

Obviously it would be easier to hold both start and stop displacements within the table but that would limit the amount of precious space available for the keys. Besides, this system is more fun to decipher! For each key, the utility first picks up the displacement of the start of the string and then calculates the next largest displacement above it. The bytes contents between those two addresses are printed out as the definition of that key. You will see that control codes such as M (RETURN) and 17 ('restore default windows') are stored as control codes – these two would be &0D and &1A respectively. It is the responsibility of the routine to decode any such bytes back into their ASCII formats. Any bytes with ASCII codes above 127 are entered by preceding the normal ASCII charater with '¦!' and these characters must be decoded as well. This feature allows you to store colour control codes and even BASIC keywords by their tokens, as in:

```
*KEY5¦! u¦! }¦! %
```

which is the tokenised form of:

```
*KEY5 REPEAT UNTIL GET
```

This only occupies three bytes in the user-key buffer!

Remember however that pressing the function key is similar to typing the command in at the keyboard, and these codes will not be recognised as BASIC keywords since you could not enter them into a BASIC program simply by typing their tokens. If you press key 5, instead of generating a wait, it just prints a '%' sign. To execute the code as BASIC is possible, but not easy.

Procedures

Only two PROCedures are required by this utility; PROCkey is called for each key to see if it has a string attached to it, and PROClistkey displays the string if one exists.

PROCkey is called with K% equal to the key number being processed (0–15) and prints out the key name and opening delimiter of the string. Having performed the calculations described above it is then in a position to call PROClistkey if it has found something there. In all cases it terminates by printing a '■' character.

Variables

K% is the key number and the program basically consists of a loop (lines 40 – 70) in which K% varies from 0 to 15. The displacement of the first byte of the string is held in ptr% and if this key turns out to be active, ptr% is incremented to point to the various bytes of the string whilst they are being printed.

PROCkey determines the end of the string by loading the lowest displacement higher than that for your key into max%, which is initially set to 255. If after all sixteen keys have been checked max% is still equal to 255, then there is no definition for that key; otherwise max% is used as the upper bound when the string is being printed.

When a key definition is found the bytes are taken one at a time from the buffer into byte% prior to PRINTing; this is necessary since the bytes have to be tested to see if they are control codes and must hence PRINT differently.

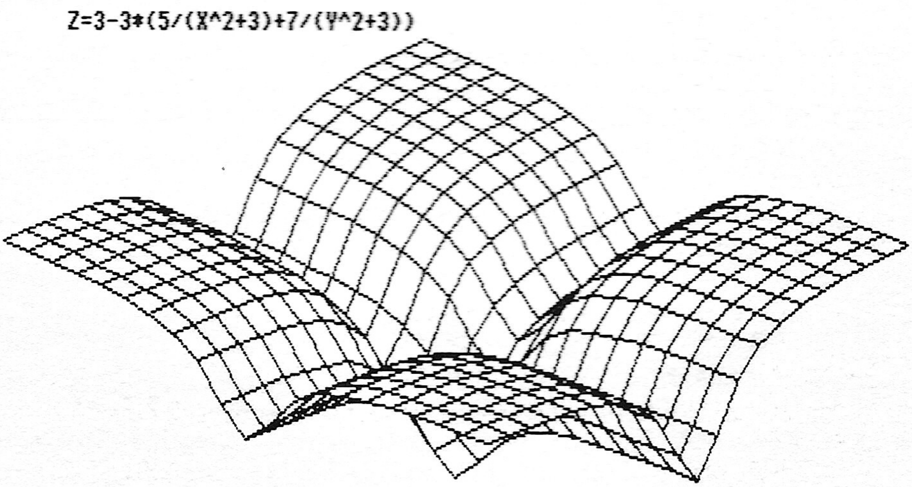
Utility 26:

Printer screen dump

Description

Most dot-matrix printers are capable of reproducing high resolution graphics provided they are supplied with the correct information, and a popular requirement for printer owners is a routine for copying the contents of the screen. This utility is just such a routine, enabling you to 'dump' the screen to an EPSON RX-80 printer. Although there are numerous different makes of printer there is now some degree of standardisation between them in, for example, their use of Escape codes, and the routine should work on other printers with the minimum of alteration. It has been used successfully with a SHINWA CP80 with no modifications at all.

The routine is in two parts: firstly a short program that assembles machine code into Page 9 and, secondly, a PROCedure that you should copy into your program and call whenever a screen dump is required. The program is presented in this way since if anything worth saving has appeared on the screen it will be as the result of a program being RUN and that program can then call the dump routine to print the screen. However many times the PROCedure is called, the first program only needs to be RUN once to assemble the machine code. A sample of the printout produced by the program appears below.



```
Z=3-3*(5/(X^2+3)+7/(Y^2+3))
```

Use

The Assembler program as given stores the machine code in Page 9, starting at &900. If this is not suitable for some reason, as when, for example, you are SPOOLing or BGETing, then the safest place is at the current value of PAGE. To do this, set PAGE=PAGE+&100 and then LOAD the Assembler routine. Change line 70 so that P% equals the old value of PAGE or, if you like, PAGE−&100, and RUN the prgram. Now LOAD the program that is to create the screen for you and merge the dump PROCedure with it. Actually, since this routine is so short, it is probably quicker to type it in – just tag it on to the end of your program. If, following the earlier suggestion, you have located the machine code at PAGE then line 30070 must be changed to:

30070 CALL (PAGE−&100)

When you want to dump the screen, call PROCdump with two parameters. The first is the starting line on the screen (0 – 31) and the second is the number of lines you require (1 – 32). For example, to dump the top half of a MODE 0 screen, you would use PROCdump (0,16).

When the PROCedure is complete, the printer is reset and then turned off. Any text or graphics windows will have been cancelled. The PROCedure may be called again at any time by your program.

```
30000 REM BASIC PROCEDURE FOR SCREEN DUM
P
30010 DEFPROCdump(P1%,P2%)
30020 VDU 23,1,0;0;0;0;
30030 ?&72=(1022-32*P1%) MOD 256
30040 ?&73=(1022-32*P1%) DIV 256
30050 ?&77=P2%
30060 VDU 26,2,1,27,1,65,1,8
30070 CALL &900
30080 VDU 1,27,1,64,3
30090 ENDPROC
30100
30110 REM 1ST PARAM=START LINE (0-31)
30120 REM 2ND PARAM=NO OF LINES (1-32)
```

```
   10 REM M/C FOR SCREEN DUMP
   20 OSWRCH=&FFEE:OSWORD=&FFF1
   30 xlo=&70:xhi=&71:ylo=&72:yhi=&73:pi
x=&74
   40 po2=&75:tot=&76:max=&77:ccs=&78
   50 ?ccs=&0A:ccs!1=&02804C1B
   60 FOR I%=0 TO 2 STEP 2
   70 P%=&900
   80 [OPTI%:
   90 .L0 LDX #0       ;start of new line-
  100 .L1 LDA #1       ;  send control cod
es for linefeed
  110     JSR OSWRCH ;  and to set 'bit-
image' mode
  120     LDA ccs,X
  130     JSR OSWRCH
  140     INX
  150     CPX #5
  160     BCC L1
  170 .L2 LDY #0       ;all codes now sent
  180     STY xlo      ;x-coordinate = 0
  190     STY xhi
  200 .L3 STY tot      ;data byte (for pri
nter) =0...so far
  210     LDA #128     ;initial power of 2
  220     STA po2
  230 .L4 LDX #(xlo MOD 256)
  240     LDA #9
  250     JSR OSWORD ;read logical colou
r of pixel...
  260     LDY #0
  270     LDA pix      ;..into pix
  280     BEQ L5       ;nothing there - so
 branch
  290     LDA tot      ;non-zero (not blac
k)
  300     ORA po2      ;so augment total
  310     STA tot
  320 .L5 SEC          ;set up location of
 next pixel down
  330     LDA ylo
  340     SBC #4
  350     STA ylo
  360     BCS L6
  370     DEC yhi
  380 .L6 LSR po2      ;divide power of 2
by 2
  390     BCC L4       ;more to do in this
 column - loop back
  400     LDA #1
  410      JSR OSWRCH
  420      LDA tot
  430      JSR OSWRCH ;send byte to print
er (only)
  440      CLC          ;set up new y-coord
inate
  450      LDA ylo
  460      ADC #32
  470      STA ylo
  480      BCC L7
  490      INC yhi
  500 .L7 CLC          ;augment x-coordina
te by 2
  510      LDA xlo
  520      ADC #2
  530      STA xlo
  540      LDA xhi
  550      ADC #0
  560      STA xhi
  570      CMP #5       ;whole line done ?
  580      BCC L3       ;no - so go back an
d do next column
  590      LDA ylo      ;yes - adjust y for
 next line down
  600      SBC #32
  610      STA ylo
  620      BCS L8
  630      DEC yhi
  640 .L8 DEC max      ;required no of lin
es done ?
  650      BNE L0       ;no - back to start
 a new line
  660      RTS          ;yes -back to calli
ng program:]
  670 NEXT
  680 END
```

How it works

Although this description applies specifically to the RX-80, it will no doubt be relevant to many other printers.

The matrix of dots used for each character is nine (wide) by nine (high) which means that the printer's most basic operation is to print a vertical column of nine dots (or blanks). A single text character is formed by printing nine such columns side by side, with the bottom dot in each column being unused in order to space out the print lines.

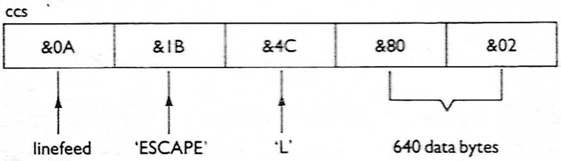
The 'bit-image mode', in which the printer can reproduce graphics, requires data to be sent to the printer in single bytes (no surprises there), with each of the eight bits controlling one dot in the column. Again the ninth position is unused. If this state of affairs continued each line of print would be separated by a blank line one dot high and the first step is to eliminate this before the dump gets started.

When the printer is first activated by the dump procedure, the print command VDU 1, 27, 1, 65, 1, 8 (or ESCape "A" 8) is used. This sets the line spacing at 8/72 inches instead of the usual 9/72 inches. Conseqiently, when a line feed is performed, the paper is wound on by eight dots instead of nine (the dots are separated by 1/72 inch) to eliminate the gap between the rows.

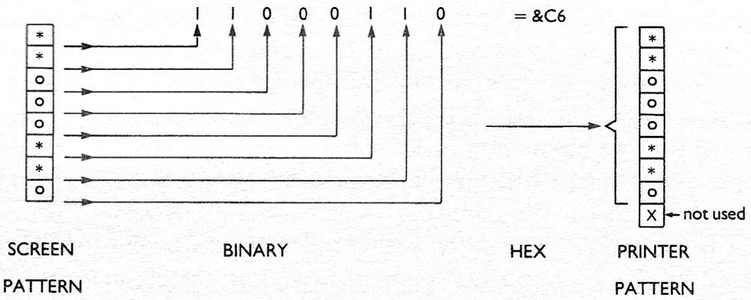
After the dump has finished, the printer is reset to the normal state so that it may be used for listings etc. by sending the command VDU 1, 64 (ESCape "@"). BASIC sets up the vertical displacement from the graphics origin of the starting line (two bytes binary) and also the number of lines to print. The machine code then takes over.

Each line is printed in exactly the same way, and the machine code consists of a loop to print each column of dots, after which the number of lines required is decremented until it hits zero, indicating that the dump has finished.

The printer mode chosen is the RX-80's 'dual-density bit image mode' which is capable of printing 960 dots across the paper. Since the BBC computer has a resolution of 640 dots horizontally the printed screen does not occupy the full width of the paper. To engage this mode we have to use the ESC L code at the start of each line and tell the printer how many dots to expect – the answer to this one is always 640. To send the information, a block of control codes is held in Zero-page and sent to the printer at the start of each line. The block looks like this:



The printer then assumes that the next 640 bytes are to be treated as dot patterns. These are read from the screen, converted to binary and passed to the printer according to the scheme:



We read the status of the dots on the screen using an OSWORD call with A=9 (see description in the *User Guide*) and if a dot is logical colour 0 (usually black) it is deemed not to be present, otherwise it is printed. In lines 230 – 390 the routine scans a column of eight dots and builds up a byte to send to the printer, so that in the diagram above it would send &C6. Having sent the byte, we augment addresses to indicate the next column of dots and proceed as before. This continues until 640 bytes have been sent, representing an entire screen line of eight dots depth. Before going on to the next line it is necessary to adjust the screen Y co-ordinate to point to the top of the next line down. If there are more lines to print, the whole process is then repeated; otherwise, the routine exits to BASIC.

Variables

The OSWORD call that reads the status of the screen pixels requires a parameter block containing the low and high X and Y co-ordinates of the point. These are held in Zero-page locations &70 to &73 and are referred to as xlo, xhi, ylo, yhi. Both ylo and yhi are set up by BASIC to point to the top pixel of the required start line and each new line starts with xlo and xhi being set to zero. The logical colour of the pixel is automatically read into &74 (clever, these OSWORD calls!) which the Assembler knows as pix. BASIC also sets up location &77 with the number of lines to print – this variable is called max.

As the Assembler looks down each column of dots, it builds up a total called tot by ORing the existing total with descending powers of 2. This power starts at 128 (=2^7) for the top dot in the column and finishes at 1 (=2^0) for the bottom one. The power of 2 in use at any time is held in po2.

At the start of each line, the block of control codes has to be sent to the printer by using the machine code equivalent of VDU 1. The first of these codes is located at ccs.

Extensions

Use of the OSWORD call for reading the screen ensures that dots whose logical colour is 0 are not printed, while all others are. If you have used the VDU 19 command to change the colour palette, then it is possible to print dots that are invisible to the naked eye (ie. black), but which are picked up by the read routine. This is not a problem, but should be borne in mind if you want to dump exotically coloured screens. The safest thing is to accept that you are printing in black-and-white and display the screen accordingly.

Alternatively, you may adjust the routine to be more selective in what it prints by replacing the simple test on pix at line 280 with something more sophisticated. While this amendment is not difficult it does seem rather illogical; the idea of a screen dump is that you get what you see. In other words, retain black backgrounds and avoid palette changes to or from black.

This utility will work in all of the graphics modes, namely 0, 1, 2, 4 and 5. Modes 3 and 6 are text only and you are unlikely to want to use the routine with those – which is just as well as the OSWORD call will not work with these modes! The teletext mode (MODE 7) is a law unto itself and is definitely not compatible with this program.

Some screen dump programs read the screen directly, rather than by an OSWORD call, and so they can handle screen MODES 0 – 6. The present system is neater and shorter (and probably a little slower), and the lack of MODES 3 and 6 is not a severe limitation. If it becomes necessary to reproduce the screen text in the middle of a program, then the PROCedure listed below will help.

```
31500 REM SCREEN TEXT DUMP
31510 DEFPROCtextdump(x%,y%)
31520 LOCAL A%,C%,X%,Y%
31530 VDU 2
31540 FOR Y%=0 TO y%
31550 FOR X%=0 TO x%
31560 VDU 31,X%,Y%
31570 A%=&87
31580 C%=(&FFFF AND USR(&FFF4)) DIV &100
31590 IF C%>32 AND C%<127 PRINT CHR$C%;
ELSE PRINT " ";
31600 NEXT X%
31610 PRINT
31620 NEXT Y%
31630 VDU 3
31640 ENDPROC
```

This PROCedure uses an OSBYTE call to read the character at the cursor position, as seen in the MODE 7 PROCedure writer utility. It should be called with two parameters: the number of columns and the number of rows (both numberings starting at 0) of the screen in the current MODE. The characters that are printed are not the characters as seen on the screen but the printer's interpretations of them, so the procedure is not a true screen dump although it does convey exactly the same information.

To prove that it works, the above listing was *not* produced by turning the printer on and typing LIST, but by listing to the screen (MODE 3, in this case) and then calling the PROCedure. Enterprising coders should have little difficulty in condensing this onto a function key.

Finally, a word about timings. Reading the screen one dot at a time is a very slow process, hence requiring the routine to be written in machine code. In keeping with the philosophy established in the String search utility, a working version was first produced in BASIC. This was extremely slow, and in comparison the printer seemed like one of the fastest peripherals ever invented.

This current version prints almost continuously and takes about 100 seconds to print a full MODE 0 screen. The RX-80 has a double speed mode (ESC "Y") but this does not allow horizontally adjacent dots to be printed and gives a much weaker image. If you wish to experiment with other printing modes (if for example you are using a different printer), the ESC "L" code is sent to the printer in line 50 of the assembler routine. It is the third hex byte (4C) of the plinged string.

Utility 27:

String sort

Description

The most common use of computers is in data-processing applications where the computer is responsible for reading and writing data files; it may also be called upon to perform some routine calculations as part of the processing. Such files can be very large (several megabytes for mainframes) and are nearly always sorted. Obviously if the data is in a random sequence it is more difficult to locate a particular piece of information within the file than if the data is ordered in some way.

A file consists of a series of 'records', each containing a set of data 'fields' which are associated with each other, and constituting a processing unit. The BBC computer – in common with most home computers – does not support such record processing since only byte, numeric and string data types may be written to or read from a file. As well as giving the string sort PROCedure, we will consider a simple way of adding a suitable record processing facility for use in conjunction with it, although each can function independently of the other. The method consists of treating strings as records and hence, in what follows, 'string' and 'record' are used interchangeably.

To sort a collection of records requires the use of one or more 'keys'. A key is a data field within the record used as the basis for sorting; up to a point, the rest of the data in the record is not important. If more than one key is used each is assigned an importance, ranging from 'major' to 'minor'. A familiar example is a set of football league tables. Here the major key is the division and the minor key is the position within it. For teams of equal standing a third key is introduced, namely alphabetical order.

The sort described here uses only one key, which may be located anywhere within the record. However, if the record is carefully structured it can give the illusion of sorting on several keys.

To sort a large number of records is a complex process, especially if all of the records cannot be loaded into the computers memory at once, and a great deal of sophisticated effort has been expended in producing efficient sorting procedures. The mathematics are complicated but the prime requirement is simple; to sort the information as fast as possible.

We look at a routine that will sort strings (remember that could mean records) based on a key starting at a fixed position within the string. The procedure expects all of the strings to be in memory at the same time, and this limits the size of the array to about 2000 strings, depending on their size and that of the program that is going to process them.

The algorithm used is not the most efficient, but as the sort is done entirely in machine code it is sufficiently fast. To illustrate the technique used, a BASIC version is given and this should help you to understand how the main procedure works.

The strings to be sorted must be held in the array A$, which can be as large as you like provided it can be held in store, and the sort will be in ascending ASCII order (if you want the strings to be in descending order you only have to read them backwards – the sort is still valid). Although the sort is a string sort, there is no reason why numeric data cannot be sorted provided it is first converted to string format. In this respect a string sort is more versatile than a numeric sort, as it simply sorts using the ASCII codes without requiring the information to be in any special format. In particular, this makes it ideal for sorting records which is one of its intended uses.

To enable the PROCedure to sort records, you should organise your data in a particular way, which is perhaps best explained by means of an example.

Suppose we have a file containing details of the members of a tennis club. For each member, the following fields are required:

Surname	12 bytes maximum
Initials	2 bytes maximum
Sex	1 byte
Membership number	3 bytes

The file could be written by using PRINT# to write each separate field as a string, but to read a record would then require four separate INPUT# commands. This is not only slightly tedious, but also illogical, since we should expect to read an entire record at one go as the record constitutes the processing unit. Furthermore, sorting such a piecemeal file would be difficult since it is not obvious which fields are associated, except by their being grouped together. We should organise the file properly and include routines to build and dismantle the records as strings.

Assume that this new information is to be inserted on the file:

Field	Data	BASIC string
Surname	Aughton	name$
Initials	J	initial$
Sex	M	sex$
Membership no.	123	number$

The record is built by executing the following code:

```
 1000  record$=name$+STRING$(12,"  ")
 1010  record$=LEFT$(record$,12)+initial
$+"  "
 1020  record$=LEFT$(record$,14)+sex$+nu
mber$
```

which creates the string "AUGHTON   J M123", and this string is written to the file as a data record. Notice that the record has been filled out with spaces to ensure that it is a fixed length. This is convenient when using discs as it enables a search to be carried out quickly, but is not so good with cassette files as extra data is being written. However, since corresponding fields occur in the same place in each record they are easier to dismantle. If we compare the above record with another:

```
rec1$="AUGHTON          J M123"
rec 2$="KIRWAN          PMF345"
```

we can see that each field starts at a fixed place within the string and can hence be used as a key for the sort.

To recreate the fields from the record, having read it in from our file, we could do something like this:

```
2000  name$=LEFT$(record$,12)
2010  initial$=MID$(record$,13,2)
2020  sex$=MID$(record$,15,1)
2030  number$=RIGHT$(record$,3)
```

This simple example introduces a method of manipulating data as records rather than a series of separate fields. Whether you choose to organise your files in this way depends on how much data they contain and the way in which you wish to access that data. A limitation of the method is that a string may only be 255 bytes in length, which is therefore the maximum size of a record. In practice, this is unlikely to be a problem.

To sort these records – or any collection of strings – you should first read them into the array A$ and then call the sort procedure. For example, to sort this file into alphabetical order by surname we simply use PROCsort(100,1) which says 'sort the first 100 strings based on a key starting at the first element of the string'. Similarly, to sort into membership number order we would use PROCsort(100,16), as that field starts at the 16th byte of the string. In this respect the machine code sort is more versatile than the BASIC (as well as being a good deal faster), since the BASIC just sorts on the whole string. Obviously, it is an easy job to amend the BASIC, but such an amendment would slow it down even more and defeat the object of explaining the sorting algorithm used.

Finally, after having sorted our strings we could then rewrite the file or process the data in some other way. It was mentioned earlier that the sort could appear to be based on several keys and in fact this is an inherent feature of the method – it comes free of charge! When the sort compares strings, it does so starting from the key field and working to the end of the record, thus taking in all the fields in sequence. Notice that if two records are otherwise equal, the shorter one is taken to be the least, which is equivalent to saying that it is padded with spaces until it is the same length as the other string. Consequently the effect of `PROCsort(100,1)` on the 'tennis' file will be to sort it into surname order and, where two surnames match, into the order of the initials. It will then place matching records into female/male order and finally membership order. This is a genuine four-key sort, the major key being surname and the minor key being membership number. To change the order of the keys, the structure of the data within the record should be altered to reflect the required order.

This feature, together with the ability to base the sort on any field of a record gives us a very powerful and versatile utility.

Use

The routine is presented as three PROCedures, which should be copied into your program and called at the appropriate times. The third procedure assembles the machine code for the actual sort and is not required thereafter; if memory is short we can take advantage of this fact to gain a few bytes. However, let us look first at the simplest way to use the utility:

1) Renumber the PROCedures with high line numbers.
2) Merge all three PROCedures with your record processing coding.
3) At a suitable point in your program call PROCass to assemble the machine code.
4) Call PROCsort.
5) Do as you will with the sorted strings – probably file them away or print them.

If you are using discs, then it is a good idea to assemble the machine code into Page 9 instead of wasting valuable space in the program area. Cassette users cannot do this as PRINT# and INPUT# use that area of memory when accessing files, so they will have to find another place for the code. In either case, considerable savings can be made by running the assembly PROCedure first and then loading in the rest of the program. This method is recommended if you have to do a large volume sort.

The sort PROCedure must be called with two parameters. The first gives the number of strings you wish to sort and the second is the starting position of the sort key within the string, The entire right-hand end of the string starting from this position is taken as the key.

If the sort PROCedure finds that something is wrong, it generates a syntax error and the explanations for these (using our line numbering) are:

Line 60: The routine cannot find the array A$ on which sort is always based.
Line 110: You are trying to sort more records than there are in the array.
Line 180: You have specified a key starting point within a string which is longer than one of your record strings.

Your program should include ON ERROR coding to deal with these situations – probably by outputting derisory comments! The time taken by the sort obviously depends on the number of strings but the following times give some indication of the speed (all timings in seconds):

Number of strings	Machine code	BASIC
100	0.37	10.41
200	1.29	39.56
300	2.81	87.0
400	4.91	153.74

Remember that the BASIC routine does not include the variable key starting position facility, and if this facility was not included in the machine code version it would be even faster. In this example, the strings sorted were titles of records (the round plastic ones!) between 3 and 28 bytes in length and the program used was:

```
10 DIM A$(400)
20 PROCass
30 X=OPENIN("SINGLES")
40 FOR I%=1 TO 400
50 INPUT#X,A$(I%)
60 NEXT I%
70 PROCsort(400,1)
80 END
```

Here are the utility listings:

```
10 REM SLOW SORT - BASIC
20 FOR J%=1 TO N
30 MIN$=A$(J%):V%=J%
40 FOR I%=J% TO N
50 IF A$(I%)<MIN$ V%=I%:MIN$=A$(I%)
60 NEXT I%
70 A$(V%)=A$(J%):A$(J%)=MIN$
80 NEXT J%
```

```
   10 REM M/C SORT (WITH OFFSET)
   20
   30 DEFPROCsort(N%,M%)
   40 size=0
   50 addr=?&482+256*?&483
   60 IF addr=0 ERROR!
   70 REPEAT
   80 IF addr!2=&03002824 PROCfound
   90 addr=?addr+256*(addr?1)
  100 UNTIL addr<PAGE
  110 IF size>N% size=N%-1 ELSE ERROR!
  120 ?&70=start MOD 256
  130 ?&71=start DIV 256
  140 ?&72=size  MOD 256
  150 ?&73=size  DIV 256
  160 ?&82=M%
  170 CALL a
  180 IF ?&82=0 ERROR!
  190 ENDPROC
  200
  210 DEFPROCfound
  220 size=addr?6+256*(addr?7)
  230 start=addr+12
  240 ENDPROC
  250
  260 DEFPROCass
  270 jalo=&70:jahi=&71:jclo=&72:jchi=&7
3
  280 ialo=&74:iahi=&75:iclo=&76:ichi=&7
7
  290 minlo=&78:minhi=&79:minal=&7A:minl
=&7B
```

```
  300 ptr=&7C:valo=&80:vahi=&81:first=&8
2
  310 DIM Z% 200
  320 FOR I%=0 TO 2 STEP 2
  330 P%=Z%
  340 [OPTI%:
  350 .a LDY #3          ;'MIN$=A%(J%)'
  360 .b LDA (jalo),Y
  370    STA minlo,Y
  380    DEY
  390    BPL b
  400    LDA jalo        ;'V%=J%'
  410    STA valo
  420    LDA jahi
  430    STA vahi
  440
  450    LDA jalo        ;'I%=J%' - by add
ress
  460    STA ialo
  470    LDA jahi
  480    STA iahi
  490
  500    LDA jclo        ;'I%=J%' - by cou
nt
  510    STA iclo
  520    LDA jchi
  530    STA ichi
  540
  550 .c LDY #3          ;get details of s
tring A$(I%)...
  560    LDA (ialo),Y    ;..its length..
  570    STA ptr+3
  580    TAX             ;into XR for big
test later
  590    DEY
  600    LDA (ialo),Y    ;..its allocated
length..
  610    STA ptr+2
  620    DEY
  630    LDA (ialo),Y    ;..its address-hi
 byte..
  640    STA ptr+1
  650    DEY
  660     LDA (ialo),Y  ;..its address-lo
 byte.
  670     STA ptr
  680
  690     CPX minl        ;is A$(I%) shorte
r than MIN$??
  700     BCC x           ;yes - use THAT l
ength
  710     LDX minl        ;no - XR=shorter
length
  720 .x CPX first        ;is the string to
o short for the chosen
  730     BCC z           ;   starting poin
t? If so, error and out
  740     LDY first       ;YR=offset into s
tring
  750     DEY
  760     INX
  770     TXA
  780     SBC first
  790     TAX             ;XR=number of byt
es to check
  800
  810 .d LDA (ptr),Y    ;compare the byte
s one at
  820     CMP (minlo),Y ;a time until:
  830     BCC e           ;      a) the firs
t is lower
  840     BNE f           ;      b) MIN$ is
still lowest
  850     INY             ;      c) they mat
ch so keep trying
  860     DEX             ;         until on
e is exhausted
  870     BNE d
  880     LDX ptr+3       ;(in this last ca
se
  890     CPX minl        ;the shorter one
is lowest)
  900     BCS f
  910
  920 .e LDA ialo        ;new min found so
 prepare to swop
  930     STA valo
  940     LDA iahi          ;implement 'V%=I%
'
  950     STA vahi
  960
  970     LDA ptr           ;this section doe
s 'MIN$=A$(I%)'
  980     STA minlo
  990     LDA ptr+1
 1000     STA minhi
 1010     LDA ptr+2
 1020     STA minal
 1030     LDA ptr+3
 1040     STA minl
 1050
 1060 .f CLC                ;implement 'NEXT
 I%' - by address
 1070     LDA ialo
 1080     ADC #4
 1090     STA ialo
 1100     BCC g
 1110     INC iahi
 1120
 1130 .g LDX iclo           ;and similarly by
 count
 1140     BNE h
 1150     DEC ichi
 1160     BMI j
 1170 .h DEC iclo
 1180     JMP c             ;'NEXT I%'
 1190
 1200 .j LDY #3             ;this bit does 'A
 $(V%)=A$(J%)'
 1210 .k LDA (jalo),Y
 1220     STA (valo),Y
 1230     DEY
 1240     BPL k
 1250
 1260     LDY #3            ;and this does 'A
 $(J%)=MIN$'
 1270 .l LDA minlo,Y
 1280     STA (jalo),Y
 1290     DEY
 1300     BPL l
 1310
 1320     CLC                ;set up 'NEXT J%'
 - by address
 1330     LDA jalo
 1340     ADC #4
 1350     STA jalo
 1360     BCC m
 1370     INC jahi
 1380
 1390 .m LDX jclo            ;and by count,too
 1400     BNE n
 1410     DEC jchi
 1420     BMI p              ;all finished - b
ack to BASIC
 1430 .n DEC jclo
 1440     JMP a              ;'NEXT J%'
 1450 .z LDX #0
 1460     STX first
 1470 .p RTS:]
 1480 NEXT
 1490 ENDPROC
```

How it works

There are two aspects to the operation of this utility; the sort routine itself and the way in which the machine code handles BASIC string arrays. The sort algorithm is by far the simpler process.

Let us consider the operation of the BASIC routine in sorting the sequence of letters (single character strings):

R E N Y A R B J

In this sort, we look through the whole list to find the 'least' data item (that having the lowest ASCII value) and fetch it to the front by swapping it with the data that was originally in that position. On our next look, we only need to start at the second item, as the first is now known to be in the correct position. Each subsequent look-through (the technical term is 'pass') is at a reduced list, gradually diminishing to only one item. Even if the data is fully sorted before the last pass is reached, the sort is unable to detect this and grinds on until the bitter end. If you sorted a pack of cards this way, you would have to go through the (diminishing) pack fifty-two times.

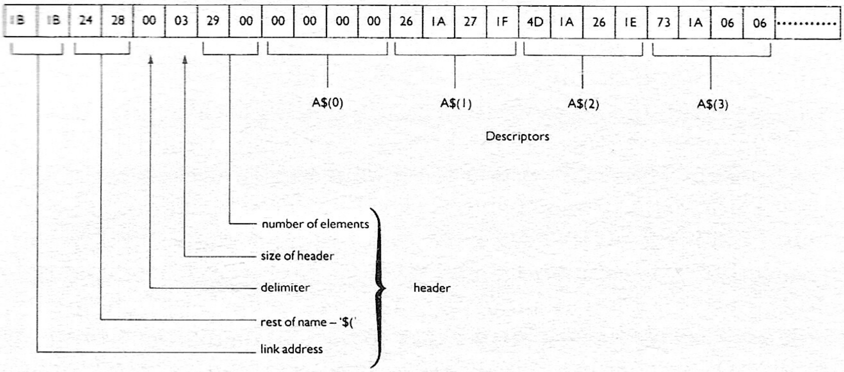
BASIC tackles the job like this: The first data item is considered to be the least so far and is stored in MINS. We now look through the list and compare each item in turn with MINS. If an item is less than MIN$ we remember its position (in V%) and give MINS the same value. When a pass has been completed we have found the true first item (it is in position V%) and it is swapped with the old first item to give the sequence:

A E N Y R R B J

The process is now repeated, but starting from the second item – the first is never seen again. Thus, to sort the list takes eight passes. Actually only seven are needed, as the last item must be in the right place if all the others are but using eight passes makes the machine code simpler. A pass requires first eight, then seven, then six, etc., comparisons to be made. One reason why this sort is inefficient is that little use is made of the information gained after each comparison has been made; other sort techniques have more to do after each comparison but have to do it less often and such sorts are usually faster. Our sort algorithm is not so bad, and is certainly a great improvement on the so-called 'bubble' sort which constantly swaps the data around since our method only requires a single swap for each pass.

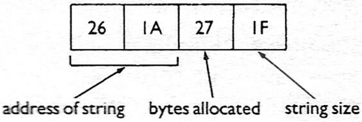
A further advantage of this method is that it produces the simplest coding, which is a great help in converting it into machine code. If you look at the machine code version you will see that it has been documented using the same terms as the BASIC coding, which it follows very closely. The big problem, of course, is that machine code does not recognise string arrays and for efficiency should be able to sort them as BASIC strings rather than copy them into an area that machine code can access, and then have to copy them back again so that BASIC can get at them. In our method the strings never get moved at all – they are simply relabelled. To understand the method we need to look at the format of a BASIC string array.

In the Symbol Table utility we looked at the BBC's method of storing variables starting from a set of pointers in Page 4. As our sort array begins with A, we start by finding the address at &482/3 which you should recall is the pointer to variables beginning with A. By following a series of link addresses, we will eventually arrive at the area allocated to the array. The entry in the variable storage area for the array will look something like this:


1B
1B
24
28
00
03
29
00
00
00
00
00
26
1A
27
1F
4D
1A
26
1E
73
1A
06
06
....
A$(0)
A$(1)
A$(2)
A$(3)
Descriptors
number of elements
size of header
delimiter
rest of name – '$('
link address
header

The use of the link address is described in the Symbol Table utility and is not relevant to our array. As the array is called A$ (. .)we would expect the rest of the name (the A is assumed, remember) to be $(which, in hex, is &2428, and to mark the end of the name it is followed by a delimiter value of &00. The next few header bytes describe the array in detail and the number of bytes this takes depends on the number of dimensions in the array. As our array has only one dimension, the header needs only three bytes, which is the significance of the &03 – if you add 3 to the address of this byte you are (almost) into the array proper.

To enable the computer to reserve space for the array details (it makes no attempt to reserve space for the strings themselves) the next two bytes indicate the size of the array. In the example just quoted we used a DIM A$(40) statement, which sets up an array of 41 elements, A$(0) to A$(40). The 41 is stored in two bytes binary (&2900 – the low byte comes first) in the array header. The next 41 sets of four bytes contain the details of each string and these bytes (let's call them a 'descriptor') have the format:


26
1A
27
1F
address of string
bytes allocated
string size

So the first two bytes point to the text of the string while the last byte indicates its length. The number of bytes allocated is not needed by us, but the machine makes use of it whenever a string is given a new value. If that value fits in the currently allocated space then it's inserted, otherwise more space has to be created for the new value. This byte does not concern us and the sort routine makes no reference to it.

If we decided to swop round some of these descriptors then the strings would still work out to be the same but they would appear (to BASIC) to have been re-ordered – in other words they would be *sorted*! For example, if in the above situation the &261A271F sequence was swapped with the &4D1A261E then BASIC would not realise that anything had gone wrong (it hasn't) but A$(1) and A$(2) would have been transposed. Thus our sorting technique consists of rearranging the descriptors; the strings that they describe can stay where they are.

Although the Assembler is written to mimic the BASIC it must be realised that a command such as A$(V%)=A$(J%) is extremely complex and not at all easy to simulate with a few bytes of machine code. However, as we have seen, we do not have to manipulate the strings at all, so things are not so bad. To make the coding easy, two versions of I% and J% are used. Firstly, there are the counts to implement the FOR . . . NEXT loops. Both counts start at the top and work backwards, which is the usual method of machine code counting. Secondly, since these counts are pretty hopeless at pointing to strings, we use an address to point at the descriptor for the current string. Thus wherever I% changes in BASIC, its machine code counterparts iclo/hi (count – going down) and ialo/hi (address – going up in fours) must be similarly changed. In the same way MINS does not exist in machine code, but a descriptor for it is held in store in Zero-page. In view of all this, the bit that does the actual comparing of strings (section d) seems pretty trivial!

When both counts have decremented to zero, we return to BASIC. If, on return, the contents of &82 have been set to zero, then an error has occurred in that a string has been found that is too short for the specified key position. Otherwise &82 holds the position of the key within the string and must be within the range 1 to (length of shortest string). Most of the free area of Zero-page is used, namely bytes &70 to &82, so you should be careful if you intend to use this sort in conjunction with other machine code routines.

At sections c and e of the code sequences of bytes are moved, and this could be done more elegantly using indexed loops, thereby saving a few bytes. Remember however that this is a sort routine – elegance doesn't come into it, especially as loop c might execute half a million times in a large sort. The extra time taken by the elegant version is quite noticeable, whereas the few bytes saved are of no great consequence. Indeed, anything that goes into the inner loop of the two nested loops in the program can have a disastrous effect on timings, whereas it is not too serious to tamper with the outer loop. An indexed loop is used in section b and causes no problems.

Variables

BASIC is given the job of finding the array A$ and addr is used as a temporary address during the search. Assuming that the array has been located, start is set up to point to the descriptors from A$ (1) onwards and size is set equal to the number of elements in the array.

The machine code uses 19 bytes in Zero-page to store important variables, including those passed to it by BASIC. These variables are: the address of the first descriptor (the one for A$ (1)) in jalo/hi, the number of strings in the array in jclo/hi and the position of the key field within the string in first. Notice that a indentifies an address while c indicates a count.

Strings are located by referring to their descriptors, two of which are held in Zero-page, with ptr holding the first byte of the descriptor for the current record (the one that BASIC calls A$ (I%)) and minlo, minhi, minal, minl representing the least string to date (MIN$ in BASIC). When a new least string is found, we remember its position by storing the address of its descriptor in valo/hi.

Extensions

This routine is complete in itself, although you may want to use it in a more efficient way as memory will be at a premium during a large sort. If possible you should divide your processing into pre- and post-sort routines where the last act of the pre-sort routine is to assemble the sort code in a safe place. Having run the sort, the post-sort section will file the sorted data away before terminating the program.

Utility 28:

Universal input routine

Description

One important requirement of all professional software is a decent input vetting routine. When the user enters data into the computer, the program should be capable of handling – without fuss –*anything* that is typed in.

Generally speaking, the INPUT command should not be used as it allows an unlimited amount of data to be entered. This can cause screen scroll and other unpleasant effects which will wreck the display. One possible solution is to use a text window for the data entry which will obviate this problem but may well introduce others.

The best way to input data is via INKEY or GET statements, as the amount and form of the data can then be strictly controlled, and the best way to achieve this is through a specialist input routine.

The operating system has an OSWORD command for this very purpose (see the description of OSWORD with A=0 in the *User Guide*) but even that has its drawbacks. INPUT uses this OSWORD call and is coded to accept all input in the ASCII range 32 to 255, i.e. all printable characters. If you use this call on a reduced range of characters then illegal items (outside your range) are not stored but still get printed on the screen. Again, this can cause scrolling.

The routine presented here comprises three short PROCedures and is a useful addition to any program that would otherwise use INPUT commands. When you call the routine, it prints out a prompt string and accepts any characters within a selected ASCII range up to a specified length limit. DELETE and RETURN will function as normal, but no other keys outside the chosen range will have any effect.

As you enter characters, the program builds a string called In$ which is eventually returned to the user's program. If that program requires numeric input, then it is your responsibility to code into the main program a statement of the form:

Number=VAL (In$)

You would then expect to perform subsequent checks on the data to ensure that it was valid – the routine only gets the data into the machine for you and cannot verify that it is exactly what you want.

By careful choice of ASCII range, much tedious vetting can be avoided. For example, specifying a range of 48 to 57 limits the characters that can be input to the range 0 to 9 so that, in this case, In$

has to represent a positive integer, which can then be processed by the host program. There is another school of thought regarding user input that says that everything entered should be accepted, and then anything incorrect specified after <RETURN> has been pressed. This is deemed to be more 'user-friendly' but seems an unnecessary elaboration. If we ensure that only strictly controlled data can be entered, the subsequent vetting of that data is greatly simplified.

Use

Merge the PROCedures with your program and then, whenever you need to accept data, call PROCinput with six parameters:

column } screen address for prompt string
row }

prompt string (can be " ")

maximum number of bytes to accept

lowest acceptable ASCII value

highest acceptable ASCII value

When you press <RETURN> the procedure exits with the input string stored in In$. A null string is not acceptable, and at least one valid character must be entered. For example take the call:

PROCinput (0, 3, "How old are you?", 2, 48, 57)

This will print the question at the start of the fourth screen line and accept either one or two characters in the range 0 – 9. While this prevents people entering 'Seventeen' (possibly quite innocently) and similar invalid items, it cannot trap '0' and it is up to your program to do any such further tests required on the input data.

One other point you should bear in mind is that you are less likely to get erroneous responses if you indicate clearly what is required. Your prompt string should, wherever possible, include guides such as '0 – 9' or 'Y/N' if the answer you expect is not completely obvious.

```
   10 REM BASIC UNIVERSAL INPUT
   20
   30 DEF PROCinput(x%,y%,prompt$,L%,Lo%
,Hi%)
   40 *FX 15,1
   50 K%=0:In$=""
   60 PRINT TAB(x%,y%)prompt$;
   70 REPEAT
   80 Z%=GET
   90 IF Z%=127 Z%=0:IF K%>0 PROCdel
  100 IF Z%>=Lo% AND K%<L% AND Z%<=Hi% P
ROCadd
  110 UNTIL Z%=13 AND K%>0
  120 ENDPROC
  130
  140 DEF PROCdel
  150 K%=K%-1:In$=LEFT$(In$,K%):VDU 127
  160 ENDPROC
  170
  180 DEFPROCadd
  190 K%=K%+1:In$=In$+CHR$(Z%):VDU Z%
  200 ENDPROC
```

How it works

Having printed your prompt string, the program goes into a loop in which it GETs bytes from the keyboard; the bytes are examined as soon as they are entered. Each byte is first tested to see if it is DELETE and, if there is already some data present, the delete operation is carried out. If a byte is outside the chosen range, or if the full quota of data has been accepted, the byte is completely ignored. The exception to this is RETURN, which ends the routine provided that some valid data has already been entered.

As bytes are added to or deleted from the string, they are printed on the screen and the string is updated. In this way, the input string reflects what is actually on the screen at any time.

Procedures

The main PROCedure is PROCinput which contains the entry and exit points for the routine. If the DELETE key is pressed, it calls PROCdel to print the delete and to recompute the input string (by chopping off the last byte added to the string).

When a valid character is received, PROCadd is called to add the character to the string and to print it on the screen.

Variables

Most of the variables used by the routine are parameters passed to it when it is called. In$ is passed back by the routine and is the string that has been input. In addition, two integer variables are used and these may be declared as LOCAL if necessary. They are Z%, which is the byte that has just been input at the keyboard, and K%, which represents the current length of the input string. K% is used to decide whether any more data – including DELETE – can be accepted.

Extensions

There are many possible variations on this routine, and it is up to the user to decide exactly what is needed. In a sense, the program presented here is itself an extension of the OSWORD routine we have already described with the 'prompt' facility added. This came about as a result of the simple observation that most data input will be in response to some query and that the coding to produce the query would be more or less the same each time. We could generalise the routine by adding more optional features, but it will obviously occupy extra memory and you will have to decide if there will be an overall benefit by including such amendments.

Bear in mind that the object of including an input routine is to make subsequent programming easier and to save duplicating code. There is no point in including such a routine, however exotic, if it is only going to be called once.

Some extra features that you may consider adding are:

1) Echo each valid (or invalid) character with a 'bleep'.

2) Clear the input area to spaces before accepting input. This enables the routine to be re-called if the data is proved to be invalid.

3) Include an error message facility so that certain types of error can be reported by the routine.

4) Allow the routine to perform range checks on the data.

5) Add more ranges so that, for example, only 0 – 9 or A – F can be input.

6) Add an edit facility whereby the cursor may be moved around the string input area to edit individual bytes. This is useful if you discover that you have made a mistake in the second byte but you are now on the 25th.

7) Let the routine print the current value of a string following the prompt so that you may edit it as in 6) above.

8) Perform partial vetting on the data. As soon as you type something disagreeable, the machine says so.

These are just some of the variations that might be included to enhance this routine. The program presented here is a basic utility which may have to be changed to suit specific requirements, although it works perfectly well as written. Certainly the concept of a specialist input routine is one you should consider for all of your programs and this utility may be used as a basis for it.

Utility 29:

Date conversion procedure

Description

Many commercial programs require extensive data vetting, and in fact such security and error-checking considerations often make up the bulk of this type of program. One common requirement is a routine to validate a date and verify that it is consistent with the day specified.

Although not strictly speaking a utility, this PROCedure is included because of its interest value and the fact that it performs a useful function – besides that, it's fun!

Given any date this century the PROCedure calculates the day of the week on which it falls. It does *not* check that the date is valid, and if you want to know what the 93rd of January is it will be quite happy to tell you. We will look at a routine to check the date later on.

The most notable feature of this routine is that it is so short; other PROCedures to do almost the same thing can be spread over a great many program lines but, as you can see, that is hardly necessary. It uses a method known to magicians and professional stage performers who can convert dates to days in about four seconds – you should be able to do so too, if you study the coding carefully and rehearse well. The technique is described by Martin Gardner in the book *Mathematical Carnival*.

Use

Simply append the PROCedure to your program and call it with the three parameters:

1. Day (1 to 31)
2. Month (1 to 12)
3. Year (0 to 99)

These parameters should be checked beforehand since the PROCedure assumes that they are correct. As written, the program just prints out the day and quits. For example, if you RUN it and enter the numbers 8, 8 and 66 the program will print MONDAY. You could suppress the printing and have it return the day in variable D$ by deleting line 50 and inserting the line:

```
135 D$=D$+"DAY"
```

If you use the routine in one of your own programs, which is its intended use, the only code you need is that in lines 80 – 160. You will probably have to RENUMBER it and you should also check that your program does not use the variables D, M, Y and D$ as they are needed by the PROCedure.

```
    10 REM DATE CONVERTER
    20
    30 INPUT"DAY,MONTH,YEAR ",D%,M%,Y%
    40 PROCday(D%,M%,Y%)
    50 PRINT'D$;"DAY"'
    60 END
    70
    80 DEFPROCday(D,M,Y)
    90 D=D+VALMID$("144025036146",M,1)
   100 D=D+(Y DIV 12)+(Y MOD 12)+(Y MOD 1
2) DIV 4
   110 IF (Y MOD 4=0) AND M<3 D=D-1
   120 RESTORE 160
   130 FOR I=0 TO D MOD 7:READ D$:NEXT
   140 ENDPROC
   150
   160 DATA "SATUR",SUN,MON,TUES,WEDNES,T
HURS,FRI
```

How it works

The algorithm is a refined version of an obvious method especially designed for fast calculation and you should refer to Martin Gardner's book for more details. This routine is just a literal translation of the method.

Variables

The PROCedure itself only uses the variables D, M, Y and D$ and they stand for . . . well, you can probably guess.

Extensions

An obvious improvement is to verify before we start that the date requested is valid and this can be simply done by including PROCvalidate (it's a pun – think about it) given here:

```
30000 DEFPROCvalidate(D,M,Y)
30010 IF Y<0 OR Y>99 error!
30020 IF M<1 OR M>12 error!
30030 X%=VALMID$("303232332323",M,1)
30040 X%=X%+28-((Y MOD 4)=0)
30050 IF D<1 OR D>X% error!
30060 IF(D=INTD)*(M=INTM)*(Y=INTY)=0 err
or!
30070 ENDPROC
```

Your coding should also include an ON ERROR address to handle the syntax errors produced in the event of a failure. Notice how the inclusion of a simple vetting routine can double the length of a piece of code! It is for this reason that we have kept error checking to a minimum in our routines – it is simply too tedious to try to catch every potential error. Of course, if any of these routines were to be used professionally, such coding would have to be included.

One further amendment would be to extend the routine to print out a calendar for a particular month or year, but in this case the coding in our routine would really only be needed once to compute a reference day from which all of the others could be calculated.

Utility 30:

Disc sector editor

Description

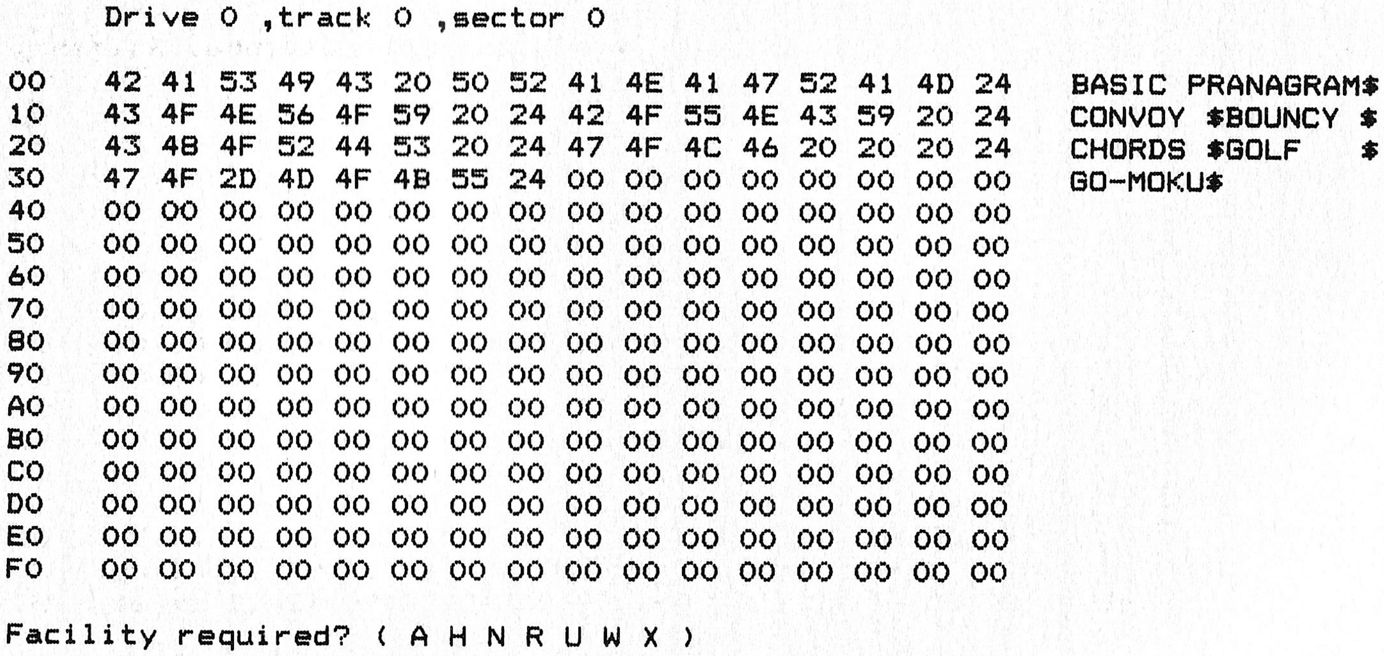
The advantages of discs over tapes are well known and it is not necessary to repeat them here. One feature of discs which makes them very easy to use is the way in which a single block of data (called a 'sector' when used in the context of discs) can be read or written. This is simply not practical with a cassette, although 'real' computers tape decks can – and do – read single blocks. They also cost several thousand pounds!

Any file (program, text, data or whatever) is written to a storage medium in blocks, and anyone who has ever loaded a cassette program into a BBC computer (that means all of us, we trust) will be familiar with the hex block count which is incremented each time a block loads. The block contains all sorts of information (see the *User Guide* on 'Cassette tape format') to enable the computer to identify the block and verify that its contents are correct. It is actually possible to read a single block by using `*OPT2,0` and a bit of deft fingerwork on the cassette controls, but it is not possible to rewrite the block in any sensible way. Actually, you can write specialised routines to write data in various formats, including a single block, but to try to insert it into the correct place on a tape would be rather hazardous, to say the least.

With a disc, there are no such problems. One of the inherent features of a disc system is the ability to read from and write to any sector (equivalent to a cassette block), and the routines to do this are included in the filing system. This utility, written entirely in BASIC, uses the OSWORD 'read/write a sector' routine and includes extensive editing facilities. If you have discs, then a utility like this one is a must.

The basic operation of the utility is to display the contents of any chosen sector of the disc in both hex and ASCII and offer you the option of editing the displayed data. Having done so, you may wish to rewrite the sector (write it back to where it came from) or write it to another place – perhaps even onto another disc.

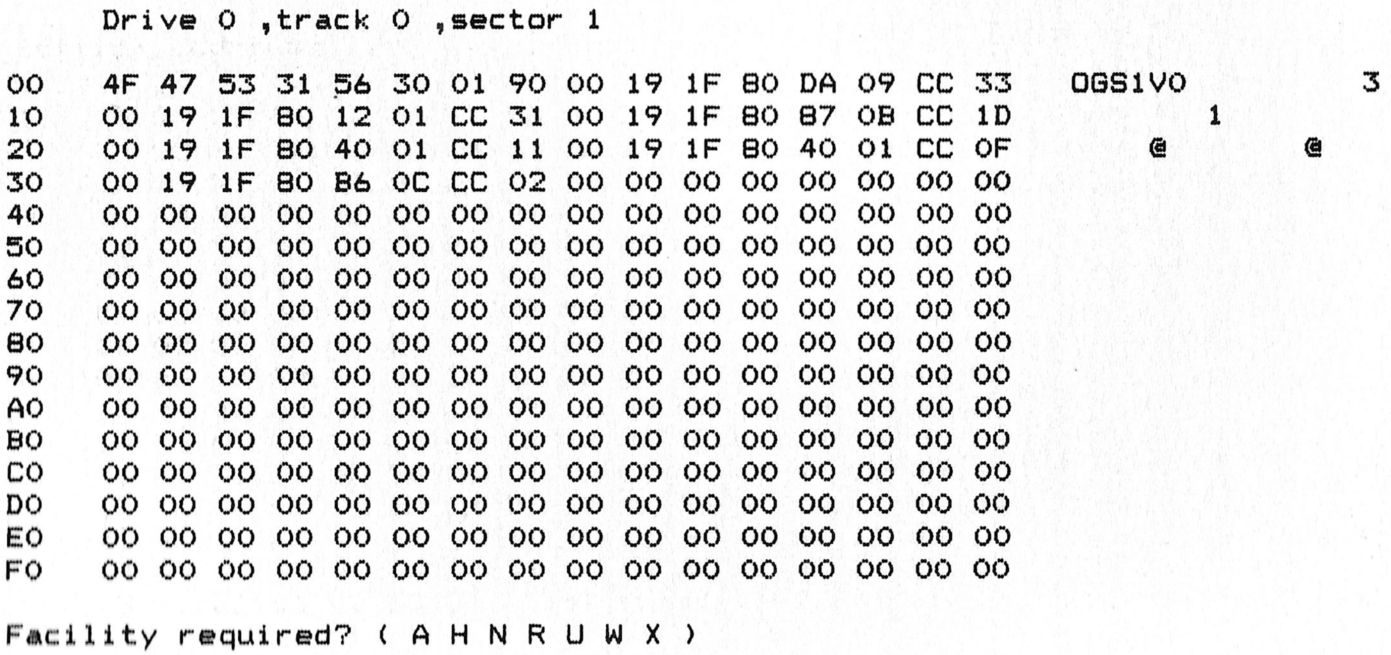
One practical use of this utility is to recover data from damaged discs or to rescue files that have been accidentally deleted. Before you do this, however, it is as well to first practise with the routine or you are likely to do more harm than good. It is also a good idea to become familiar with the layout of information on the disc – your system manual will help you here.



```
        Drive 0 ,track 0 ,sector 0

00      42 41 53 49 43 20 50 52 41 4E 41 47 52 41 4D 24    BASIC PRANAGRAM$
10      43 4F 4E 56 4F 59 20 24 42 4F 55 4E 43 59 20 24    CONVOY $BOUNCY $
20      43 48 4F 52 44 53 20 24 47 4F 4C 46 20 20 20 24    CHORDS $GOLF   $
30      47 4F 2D 4D 4F 4B 55 24 00 00 00 00 00 00 00 00    GO-MOKU$
40      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Facility required? ( A H N R U W X )
```



```
     Drive 0 ,track 0 ,sector 1

00   4F 47 53 31 56 30 01 90 00 19 1F 80 DA 09 CC 33   OGS1V0         3
10   00 19 1F 80 12 01 CC 31 00 19 1F 80 87 0B CC 1D          1
20   00 19 1F 80 40 01 CC 11 00 19 1F 80 40 01 CC 0F       @       @
30   00 19 1F 80 B6 0C CC 02 00 00 00 00 00 00 00 00
40   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Facility required? ( A H N R U W X )
```

Fig. 5. Sample screens produced by Disc Sector Editor, showing the disc catalogue.

Fig. 5. shows two sample screens from the utility and the information displayed is the catalogue (track 0, sectors 0 and 1) of a disc containing BASIC programs. MODE 3 is used so that the information will fit neatly onto the screen without scrolling and we've chosen to display the screen in cyan because we prefer it to white – change it to flashing blue and yellow at your peril!

Use

Unlike many of our other utilities this one does not need to coexist with any other program and can be simply LOADed and RUN.

Select the drive containing the test disc (consult your disc manual if you are not sure of the significance of 'Drive 0–3') and the routine will then read in the second sector. The reason for this is that this sector, whose official address is 'track 0, sector 1', contains a vital piece of information needed by the routine, namely the number of sectors on the disc – dividing this number by ten tells the utility how many tracks there are on the disc. You now enter the program proper and have the choice of options: Let us consider each option key in turn:

A: Edit ASCII. The cursor is placed at the top left hand corner of the ASCII field (the B in Fig. 5) and may be moved around the field by means of the cursor keys. The cursor movement wraps round, so that if you move off one side of the field, you will return on the other. Any ASCII data you type in (letters, numbers, punctuation) will be entered at the cursor position and the corresponding hex byte will be updated. The cursor is then stepped one place to the right.

To escape from this mode, use CTRL-X and you will return to the menu. A point of interest is that during this operation, there are four different colours on the screen at once, yet MODE 3 is a *two*-colour mode.

H: Edit Hex. The cursor is placed at the top left hand corner of the hex field and, as above, can be moved around with the cursor keys. This time, however, the movement is arranged so that you cannot land on one of the spaces between the hex items. The only acceptable input to this process are hex digits in the ranges 0 – 9 and A – F. As you amend the hex information, the ASCII display is also updated to reflect the current state of the byte.

The hex facility is more versatile than 'Edit ASCII' in that you can write data outside the printable ASCII range, and is very useful for updating addresses and file lengths in the catalogue. Return to the menu is via CTRL-X as before.

N: Read Next sector. Pressing N will cause the routine to read in the sector following the one currently on display. When you have reached the end of the disc, it will read the very first sector.

R: Read sector. Give the track and sector number you wish to read. For example, to read sector 75, we should read track 7, sector 5 (there are 10 sectors in each track. remember).

U: Update sector. Having read a sector we may wish to edit it in some way and then write it back to the disc. This option causes the currently displayed sector to be written to the currently displayed drive, track and sector address.

W: Write sector. Rather more versatile than Update, this option allows you to write the currently displayed sector to any part of the disc. Use it with care!

X: Exit. The program stops without accessing the disc again.

During read or write operations, you may get a cryptic 'Disc error' message followed by an error code number in hex. Considering the reliability of discs, such an error could be quite serious and will need looking at carefully. To convince yourself that this feature is working while you are testing out the program, swap the current disc for an unformatted one and try to read a sector. This action should give a 'disc error 18'. Hopefully, this will be the last time you ever see this message.

Experienced programmers will no doubt dream up all sorts of uses for this program, particularly in the field of disc security (both for and against, unfortunately). Like the Memory Display utility, this is one of those utilities whose usefulness grows as the skill of the user increases.

To take a simple example, let us recover a BASIC program that has just been accidentally deleted from the disc. Deleting your favourite programs is not a sensible thing to do, but it is easily done – in fact the author is something of an expert in this field. Recovery is possible as the file is not actually deleted from the disc; all that happens is that the references to it are removed from the catalogue. The recovery process will *not* work if other data has been saved to the disc since the deletion – it is at this point that the program is really lost.

Let's assume a BASIC program called HANDEL has just been deleted from the disc. To retrieve it, we must restore the catalogue information for this program. The most tedious part of the process is to locate the program on the disc. You can do this by looking through each sector, using the N facility, and, when the program is found, you should make a note of the track and sector. The program will start at the beginning of a sector. If your program includes a descriptive REM statement as the first line of your program, the task is greatly simplified.

There are better ways of locating the program, the best of course being to get the computer to do it for you, but this method is the easiest to describe. Having found the program, count how many complete sectors it occupies; or, if the end of the program is not obvious, count the number of sectors to the start of the next program. In the case of HANDEL, let's assume the start sector was track 2, sector 1 and the length was 8 sectors. The sequence of steps is now:

1) Read track 0, sector 0 using R and use A to enter the title and directory HANDEL $ just beneath the entry for GOLF.
2) Exit with CTRL-X and use U to rewrite the sector.
3) Press N to read sector 1.
4) Use H to augment the sixth byte of the sector from 30 to 38 (hex). This number is equal to 8 times the number of catalogue entries.
5) Finally, update the INFO relating to the program. This occupies 8 bytes in sector 1, in a position corresponding to the title entry in sector 0; in this example, bytes 38 – 3F (hex). The information is entered using H and will be:
   00 19 1F 80 00 08 CC 15
6) Exit with CTRL-X and update the sector.
7) Quit the utility with X – you now have your program back again!

This is a complex, though mechanical, process and if you make a habit of deleting wanted files, you should modify the program to carry out these steps automatically. A most useful inclusion would be a string search facility which located the selected string on the disc and displayed the relevent sector.

In step 5, the significance of each byte in the 8-byte INFOblock is:

BYTE	MEANING
1	load address (low byte)
2	load address (high byte)
3	execution address (low byte)
4	execution address (high byte)
5	length (low byte)
6	length (high byte)
7	bits 0, 1: very high bits of exec address bits 2, 3: very high bits of length (i.e. 0) bits 4, 5: very high bits of load address bits 6, 7: high bits of start sector
8	low byte of start sector

For BASIC programs the execution address is not important; the fact that these appear as 801F indicates that this computer is fitted with BASIC 1. The length of the program need not be exact, so it is easiest to make the first byte equal to 0 and make the second byte the length of the program in sectors, as in our example. If the program is subsequently LOADed and then re-SAVEd, the exact length will be restored in the catalogue.

Byte 7 is usually set to CC, indicating that the very high bits of the load and execution addresses are always set to 1. This does not concern us, but the last two bits do. Together with byte 8, they make up the start sector of the programe. If a program started at sector 320 (= 1 * 256 + 64) then byte 8 would hold the 64 – as hex 40, of course – and byte 7 would be incremented by 1 and stored as hex CD. In the example quoted, the start sector was 21, decimal, which in hex is 15. Consequently the last two bytes of the INFOblock are CC and 15.

If the program is in machine code, there is no way of determining the load and execution addresses except by inspired guesswork and a bit of experimentation. Although you can recover the program, there is still plenty of work to do to get it operational again.

```
   10 REM DISC SECTOR EDITOR
   20 MODE 3
   30 *FX 4,1
   40 DIM sector% 256,param% 10
   50 VDU 19,1,6,0,0,0
   60 PROCin("Which drive are you using
",0,3)
   70 drve%=ans%
   80 CLS:@%=1
   90 ?param%=drve%
  100 param%!1=sector%
  110 param%?5=3
  120 param%?9=&21
  130 trak%=0:sect%=1
  140 PROCdisc(&53)
  150 htn%=(256*(sector%?6 AND 3)+sector
%?7)/10-1
  160 REPEAT
  170 VDU 28,0,24,39,19,12
  180 PRINT "Facility required? ( A H N
R U W X ) ";
  190 F$=GET$
  200 IF F$="A" PROCedasc
  210 IF F$="H" PROCedhex
  220 IF F$="N" PROCnext
  230 IF F$="R" PROCaction(&53)
  240 IF F$="U" PROCdisc(&4B)
```

```
 250 IF F$="W" PROCaction(&4B)
 260 VDU 23,0,10,103,0;0;0;
 270 UNTIL F$="X"
 280 *FX 4,0
 290 VDU 26,12
 300 END
 310
 320 DEFPROCedasc
 330 VDU 12,28,5,17,79,2
 340 VDU 23,0,10,96,0;0;0;
 350 X%=0:Y%=0
 360 REPEAT
 370 VDU 31,50+X%,Y%
 380 F%=INKEY(0)
 390 IF F%=136 X%=(X%+15)MOD 16
 400 IF F%=137 X%=(X%+1) MOD 16
 410 IF F%=138 Y%=(Y%+17)MOD 16
 420 IF F%=139 Y%=(Y%+15)MOD 16
 430 IF F%>31 AND F%<127 PROCpatricia
 440 UNTIL F%=24
 450 ENDPROC
 460
 470 DEFPROCpatricia
 480 ptr%=16*Y%+X%
 490 VDU F%,31,3*X%,Y%
 500 PROChex(F%)
 510 sector%?ptr%=F%
 520 X%=(X%+1) MOD 16
 530 ENDPROC
 540
 550 DEFPROCedhex
 560 VDU 12,28,5,17,79,2
 570 VDU 23,0,10,96,0;0;0;
 580 X%=0:Y%=0:pos=0
 590 REPEAT
 600 VDU 31,X%,Y%
 610 F%=INKEY(0)-48
 620 IF F%=88 X%=(X%+45-pos) MOD 48:pos
=0
 630 IF F%=89 X%=(X%+51-pos) MOD 48:pos
=0
 640 IF F%=90 Y%=(Y%+17) MOD 16
 650 IF F%=91 Y%=(Y%+15) MOD 16
 660 IF (F%>=0 AND F%<10) OR (F%>16 AND
F%<23) PROCmary
 670 UNTIL F%=-24
 680 ENDPROC
 690
 700 DEFPROCmary
 710 K%=X% DIV 3
 720 ptr%=16*Y%+K%
 730 VDU F%+48
 740 IF F%>9 F%=F%-7
 750 pos=(pos+1) MOD 2
 760 X%=(X%+50-pos) MOD 48
 770 IF pos=1 F%=F%*16:MASK=15 ELSE MAS
K=240
 780 byte%=(sector%?ptr% AND MASK)+F%
 790 sector%?ptr%=byte%
 800 IF byte%<33 OR byte%>126 byte%=32
 810 VDU 31,50+K%,Y%,byte%,31,X%,Y%
 820 ENDPROC
 830
 840 DEFPROCnext
 850 sect%=(sect%+1) MOD 10
 860 IF sect%=0 trak%=(trak%+1) MOD (ht
n%+1)
 870 PROCdisc(&53)
 880 ENDPROC
 890
 900 DEFPROCaction(fac%)
 910 VDU 28,0,24,39,18,12
 920 PROCin("Track ",0,htn%)
 930 trak%=ans%
 940 PROCin("Sector ",0,9)
 950 sect%=ans%
 960 PROCdisc(fac%)
 970 ENDPROC
 980
 990 DEFPROCdisc(Z%)
1000 param%?6=Z%
1010 param%?7=trak%
1020 param%?8=sect%
1030 param%?10=0
1040 A%=&7F
1050 X%=param% MOD 256:Y%=param% DIV 25
6
```

```
 1060 CALL &FFF1
 1070 IF param%?10 PROCerror
 1080 IF Z%=&53 PROCdisplay
 1090 ENDPROC
 1100
 1110 DEFPROCerror
 1120 CLS
 1130 PRINT"Disc error ";~param%?10
 1140 VDU 7
 1150 T%=TIME:REPEAT UNTIL TIME>T%+300
 1160 ENDPROC
 1170
 1180 DEFPROCdisplay
 1190 VDU 26:ptr%=0
 1200 PRINT "    Drive "drve%" ,track "
trak%" ,sector "sect%" "'
 1210 FOR Y%=0 TO 240 STEP 16
 1220 PROChex(Y%)
 1230 PRINT "  ";
 1240 ascii$=""
 1250 FOR X%=4 TO 49 STEP 3
 1260 byte%=sector%?ptr%
 1270 IF byte%>32 AND byte%<127 A$=CHR$b
yte% ELSE A$=" "
 1280 ascii$=ascii$+A$
 1290 PROChex(byte%)
 1300 ptr%=ptr%+1
 1310 NEXT
 1320 PRINT "  ";ascii$
 1330 NEXT
 1340 ENDPROC
 1350
 1360 DEFPROChex(Z%)
 1370 IF Z%<16 PRINT"0";
 1380 PRINT ~Z%;" ";
 1390 ENDPROC
 1400
 1410 DEFPROCin(A$,min%,max%)
 1420 REPEAT
 1430 VDU 7
 1440 PRINT'A$;"("STR$(min%)"-"STR$(max%
)")";
 1450 INPUT ans%
 1460 UNTIL ans%>=min% AND ans%<=max%
 1470 ENDPROC
```

How it works

This utility is written in classic 'menu-driven' style in that it consists of a short loop, from whence one of several procedures is called to perform the selected option. On return from the procedure, the loop is re-entered – the only way out of the loop is to type X.

The disc read and write operations are performed by the short PROCdisc, whose action is described later; for now, all we need to know about this routine is that it can read any sector to, or write any sector from, an area of memory whose first byte has the address sector%. If the screen data is edited the information in this area is also updated so that it reflects the state of the screen at all times.

Because the operation of this program is more or less determined by the PROCedures within it, the program is best understood by looking at each in turn.

Procedures

PROCedasc is called to edit ASCII bytes (option A). It sets up a text window around the ASCII field and creates a block cursor. The PROCedure mainly consists of a loop looking for the user to press a cursor key (note the neat use of MOD to provide wrap-around movement), a key in the ASCII range, or CTRL-X to exit. If a valid ASCII item is to be entered, PROCpatricia is called. (If Acorn can name parts of their computer after people, I can do the same with my programs!) This prints the byte, updates the hex display and enters the information into the internal copy of the screen located at sector %.

PROCedhex is similarly used to edit hex bytes. The movement is slightly more complicated, since a new hex byte is two digits wide and the bytes are separated by spaces. When a valid digit is entered PROCmary is called to update the various affected areas.

PROCnext is a simple procedure to read the next sector. It calls PROCdisc – which is not so simple – to do the hard work.

PROCaction is called whenever a random access is required. The parameter is set to &53 to read and to &4B to write. Having accepted the track and sector numbers within the correct ranges it, too, calls PROCdisc to perform the operation.

PROCdisc. This is called with the same parameter as PROCaction, and its function is to complete the parameter block required by OSWORD before calling OSWORD with A=&7F. The details of this operation are given in the disc manual but for completeness we should look at the composition of the parameter block. It is stored at param% (its first byte) and contains these 11 bytes:

Displacement	Contents
0	Drive number
1–4	Address of sector%
5	Number of parameters (3)
6	Command: &53 to Read, &4B to Write
7	Track
8	Sector
9	Length (&21 for one sector)
10	Error code returned by OSWORD (0)

If a non-zero error code is returned, PROCerror is called to display it. Following a read command, PROdisplay is called to update the display.

PROCdisplay. Once a new sector has been read into the reserved area, the bytes are read and displayed on the screen. The method used is very similar to that employed in the Memory Display utility.

PROChex prints out its input parameter in hex and adds a leading zero if necessary.

PROCin has three parameters: a prompt string, and the least and greatest acceptable values. It announces itself with a bleep and prints the prompt string, followed by a range. On exit, ans% will be the input response in that range.

Variables

The current drive, track and sector are stored in drive%, trak% and sect% respectively. These variables uniquely describe the sector on display at any time.

The OSWORDcommand which accesses the disc requires an eleven-byte parameter block, starting at param%, and an area to store the sector, which is located at sector%. To access the various bytes in the sector a pointer is used as a displacement from the first byte. This pointer is calculated from the screen address of the cursor (held in X% and Y%) and is referred to as ptr%.

When the utility is first entered, the number of sectors on the disc is read from the 7th and 8th bytes of sector 1, where it is stored by the formatting process, and this is then converted to htn% which stands for 'highest track number'. In the case of 40-track discs, this will be equal to 39, since the numbering starts at 0.

Most of the other variables used by the routine are 'local' to a particular area, although they are not defined as such. While their use may not be self-explanatory, you do not have to look very deeply to see how they are used. In common with other utilities in the book, whenever an input is received from the keyboard to select a facility, the byte is referred to as F$ or F%, depending on its format.

Extensions

As this program allows you to read or write single sectors of the disc it can be made to perform many different functions. One suggestion has been to automate the process for recovering lost files and this is certainly possible; the only major extras being a string search feature and an extra facility – how about S for 'Salvage'?

Another possibility is the addition of a second catalogue on the disc. One criticism of the BBC disc system is the fact that only 31 files can be held on the disc. This limit is derived from the catalogue capacity, not the disc itself; so that by introducing a second catalogue, 60 files could be held. One file would have to be a special 'catalogue swapping' programme designed to update and adjust the catalogues as data is added to and deleted from the disc. Such an amendment qualifies as a project in its own right rather than a modification to the Disc Sector Editor. However, this utility is a good place to start and some of the read/write routines within it would certainly be useful for that project.

Utility 31:

Multiple precision arithmetic

Description

If you are a child of the calculator age, it is likely that you have only a nodding acquaintance with 'long' multiplication and division. For better or for worse, these skills have almost been made redundant by the invention of new calculating aids and, whilst most people can perform basic number manipulation, the underlying principles are less well understood. Unless you 'know your tables' the methods are long-winded and prone to error.

Most everyday numbers are small enough to fit on the cheapest pocket calculator and few of us ever venture into the realms of numbers that are so large that they can only be computed by laborious pencil and paper methods.

However, large numbers do have their uses. In financial programs the fact that numbers can only be held to 10 significant figures, as on most home micros – some handle less figures, a very few handle more – is a severe limitation. Given that the last figure will be unreliable anyway and two digits are needed for the pence, the greatest total that can safely be used is only in tens of millions of pounds. This may seem like a lot (it is a lot!) but it does mean that you cannot use a micro to run a bank or a large business unless this problem has been overcome.

Of course, larger computers can support 20 or 30 digit accuracy which is enough for all financial and most scientific purposes. If it is necessary to use numbers bigger than this, then specialised routines must be written to deal with them, and one such routine is the subject of this final utility. The sole justification for dealing with such large numbers is the interest of the numbers themselves, rather than what they might represent. This field, admittedly not everyone's cup of tea, involves Number Theory, which is one of the most obtuse areas of mathematics. However, there is nothing deep or cerebral about this particular program, and it can be used at a very simple level to manipulate numbers great and small.

The routine can add, multiply, subtract or divide two positive integers provided the result is less than 255 digits long. It is written as a PROCedure and, by repeatedly calling it, you can produce powers, factorials and even (if you know your maths) roots of numbers. This application is a 'fun' use of the routine – a practical use would be in a program that handled large amounts of money.

Such programs have to manipulate data correct to the very last digit.

The answer is produced as a string ready for printing or for return to the PROCedure for a bit more treatment. If the result of a subtraction is negative, then the result is returned as a positive number and a flag is set (the routine also bleeps) for you to test and then act on as appropriate. If you attempt to divide a number by a larger number, the routine exits with the flag set to indicate that this has happened. Otherwise, the result of a division is the integer part of the answer, that is, it behaves like the BASIC command DIV.

The program is written in BASIC and, since it treats numbers as being composed of single digits (fair does – they *are*) it is fairly slow, especially at division.

Use

The set of procedures should be merged with the program that is to use them. If you only want to perform one or two types of calculations, then great chunks of the routine can be omitted – the table below shows which PROCedures can be left out if you do not need a particular facility.

Facility	Operation	Routines to be omitted
1	Addition	add
2	Multiplication	mult
3	Subtraction	subtract, negative
4	Division	divide, isBgreater, shiftleft, shiftright

The host program should dimension three integer arrays, namely A%(255), B%(255), and C%(255). These are needed by the routine to hold partial results while it is working. A$ and B$ should be set up with the two integers to be processed and, in a subtraction or a division, you should ensure that VAL (A$) > VAL (B$). Call the routine with a facility number in the range 1 – 4 as the parameter, and on exit C$ will be set up with the answer. If, on return, the flag N% is non-zero, then something unusual has happened; if N%=-1 then you are trying to divide a number by a larger one (solution: tag a few 0's onto it), and if N%=1, the result of a subtraction should be thought of as negative.

As a simple illustration, the second listing given here is a suitable host program for the routine. It invites you to type in two numbers and to then select one of the four operations. After calling the utility, it prints the answer and then asks for more numbers.

```
30000 DEFPROCfermat(F%)
30010 PROCinitial
30020 IF F%=1 PROCadd
30030 IF F%=2 PROCmult
30040 IF F%=3 PROCsubtract
30050 IF F%=4 PROCdivide
30060 PROCreorderc
30070 IF N% VDU 7
30080 ENDPROC
30090
30100 DEFPROCinitial
30110 N%=0:LA=LENA$:LB=LENB$
30120 IF LA>LB L%=LA ELSE L%=LB
30130 FOR I%=0 TO L%
30140 A%(I%)=-(I%<LA)*VALMID$(A$,LA-I%,1
)
30150 B%(I%)=-(I%<LB)*VALMID$(B$,LB-I%,1
)
30160 NEXT
30170 ENDPROC
30180
30190 DEFPROCadd
30200 U%=L%
30210 PROCzeroc
30220 FOR I%=0 TO L%-1
30230 C%(I%)=A%(I%)+B%(I%)
30240 NEXT
30250 ENDPROC
30260
30270 DEFPROCmult
30280 U%=2*L%
30290 PROCzeroc
30300 FOR I%=0 TO L%-1
30310 FOR J%=0 TO L%-1
30320 K%=I%+J%
30330 C%(K%)=A%(J%)*B%(I%)+C%(K%)
30340 NEXT J%,I%
30350 ENDPROC
30360
30370 DEFPROCsubtract
30380 U%=L%-1
30390 PROCzeroc
30400 C%=0
30410 FOR I%=0 TO L%-1
```

```
30420 K%=A%(I%)-B%(I%)-C%
30430 IF K%<0 K%=K%+10:C%=1 ELSE C%=0
30440 C%(I%)=K%
30450 NEXT
30460 IF C% PROCnegative
30470 ENDPROC
30480
30490 DEFPROCnegative
30500 N%=1
30510 FOR I%=0 TO L%-1
30520 C%(I%)=9-C%(I%)
30530 NEXT
30540 C%(0)=C%(0)+1
30550 ENDPROC
30560
30570 DEFPROCdivide
30580 PROCisBgreater
30590 IF yes N%=-1:ENDPROC
30600 L%=-1
30610 REPEAT
30620 L%=L%+1
30630 PROCshiftleft
30640 PROCisBgreater
30650 UNTIL yes
30660 U%=L%
30670 PROCzeroc
30680 PROCshiftright
30690 C%=0
30700 FOR I%=0 TO LB
30710 K%=A%(I%)-B%(I%)-C%
30720 IF K%<0 K%=K%+10:C%=1 ELSE C%=0
30730 A%(I%)=K%
30740 NEXT
30750 C%(L%)=C%(L%)+1
30760 PROCisBgreater
30770 IF NOT yes GOTO 30690
30780 PROCshiftright
30790 L%=L%-1
30800 IF L%>-1 GOTO 30760
30810 ENDPROC
30820
30830 DEFPROCisBgreater
30840 K%=LA+1
30850 REPEAT
```

```
30860 K%=K%-1
30870 UNTIL K%=0 OR B%(K%)<>A%(K%)
30880 yes=B%(K%)>A%(K%)
30890 ENDPROC
30900
30910 DEFPROCshiftleft
30920 FOR I%=LB TO 1 STEP -1
30930 B%(I%)=B%(I%-1)
30940 NEXT
30950 B%(0)=0:LB=LB+1
30960 ENDPROC
30970
30980 DEFPROCshiftright
30990 B%(LB)=0:LB=LB-1
31000 FOR I%=0 TO LB
31010 B%(I%)=B%(I%+1)
31020 NEXT
31030 ENDPROC
31040
31050 DEFPROCzeroc
31060 FOR I%=0 TO U%
31070 C%(I%)=0
31080 NEXT
31090 ENDPROC
31100
31110 DEFPROCreorderc
31120 C%=0:CSIG=0:C$=""
31130 FOR I%=0 TO U%
31140 K%=C%(I%)+C%
31150 C%(I%)=K% MOD 10:C%=K% DIV 10
31160 IF C%(I%) CSIG=I%
31170 C$=CHR$(48+C%(I%))+C$
31180 NEXT
31190 C$=RIGHT$(C$,CSIG+1)
31200 ENDPROC
```

```
10 DIM A%(255),B%(255),C%(255)
20 A$=STRING$(255," "):B$=A$:C$=A$
30 REPEAT
40 INPUT "A="A$
50 INPUT "B="B$
```

```
   60 INPUT "1:A+B  2:A*B  3:A-B  4:A/B
"Op%
   70 IF Op%<5 PROCfermat(Op%):PRINT'C$'
   80 UNTIL Op%>4
   90 END
```

How it works

The incoming numbers are extracted from their strings one digit at a time and placed into two arrays. The facility chosen then determines what happens to the arrays. Very simply, all that happens is that the usual techniques of addition, multiplication, etc., are applied to the digits to produce the result, also in array form. Finally, the digits are strung together again to provide the answer in string form.

As the computer is dealing with single digits, the largest calculation that it has to worry about is 9 * 9, which is unlikely to tax a machine of the BBC's powers. The catch is of course that many such calculations have to be performed and it is this factor which slows down the operation. A much faster technique would be to split the numbers up into 'digits' which were actually 5 (this is risky, 4 would be better) decimal digits long and work with those. The penalty for this improvement is that the code would have to be a great deal more complicated.

The method used for division is rather complex and relies on a process of shifting the decimal point (multiplying and dividing by 10) and repeated subtraction. It is exactly analagous to the pencil-and-paper technique of 'long' division. The other operations are easier to understand, so let us consider the simplest case, namely addition.

Suppose we require to perform the sum 67853 + 1729, the arrays A% and B% would need to be set up like this:

	5	4	3	2	1	0
A%	0	6	7	8	5	3
B%	0	0	1	7	2	9

In addition the first 6 elements of the array C%will be zeroised – this is the area in which the result is stored. Notice that each number has at least one leading zero (this is convenient if it is required to shift the numbers in a division) and that the numbers are both the same length. If the numbers are offered to the routine with leading zeros, these are stripped off and the larger number will be left with just one zero at the beginning.

If you study the addition procedure, you will see that the array C% is then computed giving:

C%(5)=0
C%(4)=6
C%(3)=8
C%(2)=15
C%(1)=7
C%(0)=12

When the array C% is converted to a string, all of the 'carries' are done before each digit is added to the string. For example, since C%(0)=12 the final digit of the string (the least significant) will be 2 and C%(1)will receive a carry, making it equal to 8. Before returning the string, the procedure makes sure that all leading zeros have been stripped off and the final result will then be C$="69582".

Multiplication and subtraction should also be easy to follow. A flow-chart showing the algorithm used by the division procedure is given in Fig. 6 opposite.

Procedures

The PROCedure containing entry and exit points for the routine is called fermat in honour of one of the greatest of all mathematicians, who was rather adept at handling large numbers. This PROCedure calls three others: PROCinitial to build up arrays A% and B%, one of the 'facility' PROCedures already mentioned, and finally PROCreorderc to restore the array C% to string form. While the facility is being performed, PROCzeroc will be called to set sufficient elements of array C% to zero – the answer will occupy those zeroised places.

If a subtraction has given a negative result, then PROCnegative is called to '10's complement' the answer and to indicate the fact that this has happened by setting the flag N%.

PROCshiftleft and PROCshiftright are used by the division routine to perform those actions on the array B%, and PROCisBgreater is used by the same routine to determine whether the number represented by array B% is greater than that represented by array A%.

Variables

Most of the important variables have already been discussed; others of interest are LA, LB and L%. These are, respectively, the true lengths of A$ and B$ as initially received by the routine and then the greater of these two. The division procedure needs to adjust LB as it shifts B% around and it also needs to work out its own value of L%.

U% is the expected maximum length of the result, and C% is always used to represent a carry (it could be greater than 1). The variable yes answers the question 'Is B$ greater than A$'.

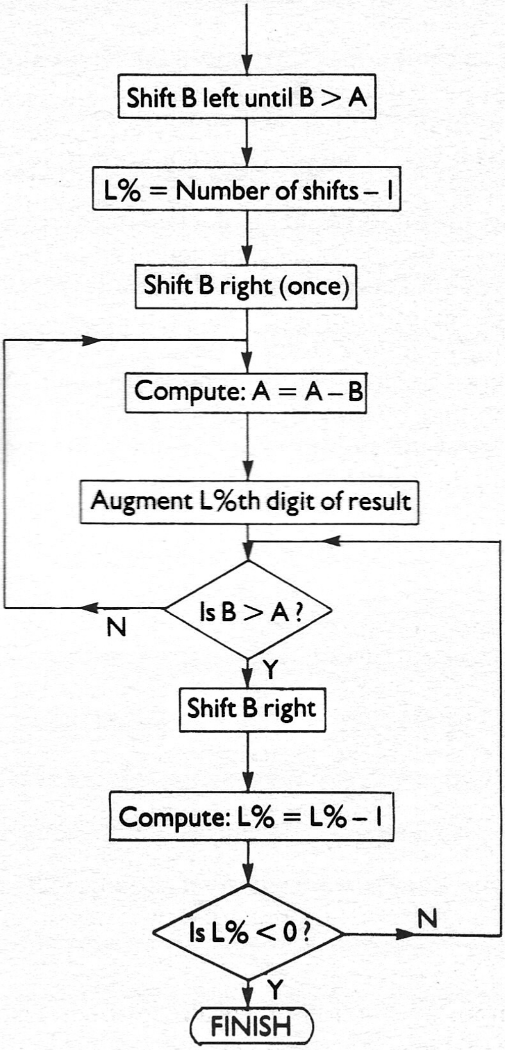

Shift B left until B > A
L% = Number of shifts – 1
Shift B right (once)
Compute: A = A – B
Augment L%th digit of result
Is B > A ?
N
Y
Shift B right
Compute: L% = L% – 1
Is L% < 0 ?
N
Y
FINISH

Fig. 6. Flowchart to show routine to divide A by B, using 'long' division.

Extensions

The most likely reason for amending this program is to increase the speed, since it can be rather slow in multiplying or dividing gigantic numbers. This is not a fault in the program (considering the size of the numbers involved it often seems remarkable that the computer can do the sum at all), but it may turn out to be a limitation, especially if you are going to use it to work out powers or factorials. In fact, the actual calculations performed by the program are so straightforward that it could even be converted to machine code without too much trouble.

Alternatively, the digits could be blocked in groups of four, instead of singly, which would mean less calculation would be needed, again giving an increase in speed. A further benefit is that bigger numbers could be handled, but printing them could be a problem as the maximum string length is 255 characters.

If it is required to deal with numbers greater than 255 digits in length, they should be stored in a string array, rather than in the single string C$ – at this point the whole program is beginning to get out of hand and it should be replanned for the specific task you have in mind. The existing procedures should form a good basis for any extensions that you may wish to apply.

Robert Erskine & Humphrey Walwyn with Paul Stanley and Michael Bews
Sixty Programs for the Sinclair ZX Spectrum £5.95

Sixty Programs for the BBC Micro £5.95

Sixty Programs for the Dragon 32 £5.95

Sixty Programs for the Oric 1 £5.95

Sixty Programs for the Atari £5.95

Sixty Programs for the Commodore 64 £5.95

Sixty Programs for the Vic 20 £5.95

Sixty Programs for the Electron £5.95

Ian Adamson
The Companion to the Oric 1 £5.95

Geoff Wheelwright
The Companion to the BBC Micro £4.95

Keith Bowden
The Companion to the Commodore 64 £5.95

Jeremiah Jones and Geoff Wheelwright
The Companion to the Electron £5.95

All these books are available at your local bookshop or newsagent, or can be ordered direct from the publisher. Indicate the number of copies required and fill in the form below

Name____
(Block letters please)

Address______________________________________

____

Send to Pan Books (CS Department), PO Box 40, Basingstoke, Hants. Please enclose remittance to the value of the cover price plus: 35p for the first book plus 15p per copy for each additional book ordered to a maximum charge of £1.25 to cover postage and packing. Applicable only in the UK.

While every effort is made to keep prices low, it is sometimes necessary to increase prices at short notice. Pan Books reserve the right to show on covers and charge new retail prices which may differ from those advertised in the text or elsewhere.

Jean Frost
Instant Arcade Games for the Sinclair ZX Spectrum £3.95

Instant Arcade Games for the BBC Micro £3.95

Instant Arcade Games for the Dragon 32 £3.95

Instant Arcade Games for the Electron £3.95

J. J. Clessa
Micropuzzles £2.95

Ian Scales
Spectrum Peripherals Guide £4.95

Tony Takoushi
Best Software Guide: Vic 20/Commodore 64 £3.95

Best Software Guide: Spectrum Games £3.95

Jeff Aughton
Invaluable Utilities for your BBC Micro £5.95

All these books are available at your local bookshop or newsagent, or can be ordered direct from the publisher. Indicate the number of copies required and fill in the form below.

Name____
(Block letters please)

Address____

____

Send to Pan Books (CS Department), PO Box 40, Basingstoke, Hants. Please enclose remittance to the value of the cover price plus: 35p for the first book plus 15p per copy for each additional book ordered to a maximum charge of £1.25 to cover postage and packing. Applicable only in the UK.

While every effort is made to keep prices low, it is sometimes necessary to increase prices at short notice. Pan Books reserve the right to show on covers and charge new retail prices which may differ from those advertised in the text or elsewhere.

THE ULTIMATE PROGRAMMER'S TOOLKIT – INVALUABLE PROGRAMMING AIDS FOR YOUR COMPUTER!

Utilities to take the pain out of programming . . .
Utilities to explore the hidden depths of your Beeb . . .
Utilities to customise your machine and upgrade your programming capacity.

In this collection of powerful tools for the serious programmer, Jeff Aughton provides the complete software toolkit needed to construct efficient programs, and the weapons you require to disassemble them and annihilate the bugs. The accompanying text fully explains the use and mechanics of each utility and in so doing provides readers with a high-level tour of the inner world of the BBC Micro.

Among the utilities included

✻ A Disassembler – delving deeper into the machine code ROMs of your Beeb

✻ A Music Processor – teach your micro to beep in tune

✻ Bad Program Fix – resurrect your fatally wounded code

✻ Sorts – efficiently order your indexes and files with this nippy little routine

✻ Teletext Picture Editor – create your own Prestel frames with this innovative graphics system.

0 330 28480 0

U.K. £5.95

INVALUABLE UTILITIES for the BBC MICRO

Pan
PCN